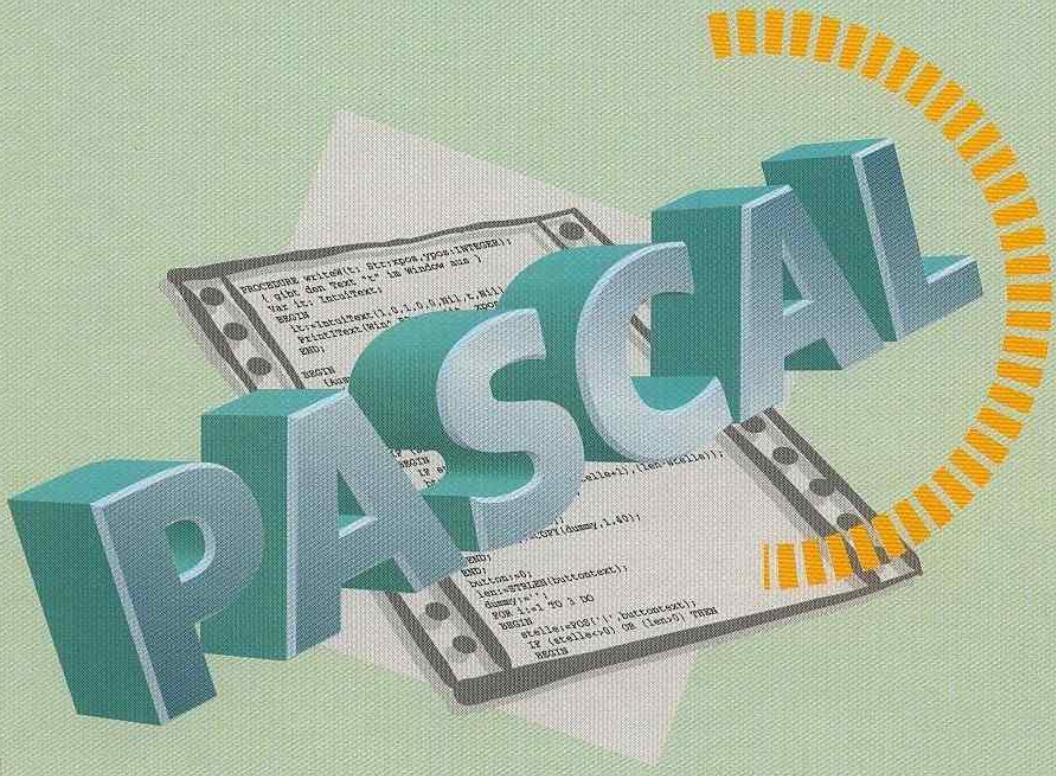
Maxon PASCAL 3

Integriertes Pascal-Compilersystem



AMIGA®

MAXON computer

FUTURE

AMIGA

MaxonPASCAL

Integriertes Pascal-Compilersystem

MAXON
computer

MaxonPASCAL 3

Autoren:

Compiler: **Jens Gelhar**
Editor: **Thomas E. Wieger**
Units/Includes: **Jürgen Schmitz**

© 1989-94 MAXON Computer GmbH

Alle Rechte vorbehalten. Dieses Handbuch und die dazugehörige Software ist urheberrechtlich geschützt. Es darf in keiner Form (auch auszugsweise) mittels irgendwelcher Verfahren reproduziert, gesendet, vervielfältigt bzw. verbreitet oder in eine andere Sprache übersetzt werden.

Bei der Erstellung des Programmes, der Anleitung sowie Abbildungen wurde mit allergrößter Sorgfalt vorgegangen. Trotzdem können Fehler nicht ausgeschlossen werden. MAXON übernimmt keinerlei Haftung für Schäden, die auf eine Fehlfunktion von Programmen zurückzuführen sind.

Alle Informationen, die in der vorliegenden Anleitung enthalten sind, werden ohne Rücksicht auf einen eventuellen Patentschutz veröffentlicht. Ebenso werden Warenzeichen ohne Gewährleistung einer freien Verwendung benutzt.

Support

Bei Fragen zu **MaxonPASCAL 3** wenden Sie sich bitte an:

MAXON Computer GmbH
"**MaxonPASCAL 3** Support"
Postfach 5969
65734 Eschborn

Copyrights und Warenzeichen:
Commodore® und Amiga® sind eingetragene Warenzeichen der Commodore-Amiga Inc.
AmigaDOS™, Kickstart™, Intuition™ und Workbench™ sind Warenzeichen der Commodore-Amiga Inc.
Amiga Workbench™ ist ein Copyright der Commodore Amiga Inc.
MAXON® ist ein eingetragenes Warenzeichen der MAXON Computer GmbH

MAXON Software-Lizenzbedingungen

1 Allgemeines

Gegenstand dieses Vertrages ist das Benutzungsrecht für MAXON Computerprogramme, für die Benutzungsanleitung sowie für sonstiges zugehöriges schriftliches Material, nachfolgend zusammenfassend als Software bezeichnet.

2 Vervielfältigungsrechte

(1) Der Anwender darf das gelieferte Programm vervielfältigen, soweit die jeweilige Vervielfältigung für die Benutzung des Programms notwendig ist. Zu den notwendigen Vervielfältigungen zählen die Installation des Programms vom Originaldatenträger auf den Massenspeicher der eingesetzten Hardware sowie das Laden des Programms in den Arbeitsspeicher.
(2) Darüber hinaus kann der Anwender eine Vervielfältigung zu Sicherungszwecken vornehmen. Es darf jedoch jeweils nur eine einzige Sicherungskopie angefertigt und aufbewahrt werden. Diese Sicherungskopie ist als solche des überlassenen Programms zu kennzeichnen.
(3) Weitere Vervielfältigungen, zu denen auch die Ausgabe des Programmcodes auf einen Drucker sowie das Fotokopieren des Handbuchs zählen, darf der Anwender nicht anfertigen.

3 Mehrfachnutzungen und Netzwerkeinsatz

(1) Der Anwender darf die Software auf jeder ihm zur Verfügung stehenden Hardware einsetzen. Wechselt der Anwender jedoch die Hardware, muß er die Software vom Massenspeicher der bisher verwendeten Hardware löschen. Ein zeitgleiches Einspeichern, Vorrätighalten oder Benutzen auf mehr als nur einer Hardware ist unzulässig.
(2) Der Einsatz der überlassenen Software innerhalb eines Netzwerkes oder eines sonstigen Mehrstations-Rechensystems ist unzulässig, sofern damit die Möglichkeit zeitgleicher Mehrfachnutzung des Programms geschaffen wird. Möchte der Anwender die Software innerhalb eines Netzwerkes oder sonstiger Mehrstations-Rechensysteme einsetzen, muß er eine zeitgleiche Mehrfachnutzung durch Zugriffsschutzmechanismen unterbinden oder an den Lieferanten eine besondere Netzwerkgebühr entrichten, deren Höhe sich nach der Anzahl der an das Rechensystem angeschlossenen Benutzer bestimmt.
(3) Die im Einzelfall zu entrichtende Netzwerkgebühr wird dem Anwender von dem Lieferant umgehend mitgeteilt, sobald der Anwender dem Lieferanten den geplanten Netzwerkeinsatz einschließlich der Anzahl angeschlossener Benutzer schriftlich bekanntgegeben hat. Unsere Anschrift ist dem Benutzerhandbuch zu entnehmen. Der Einsatz im Netzwerk ist erst nach der vollständigen Entrichtung der Netzwerkgebühr zulässig.

4 Weiterveräußerung

(1) Der Anwender darf die Software einschließlich des Benutzerhandbuchs und des sonstigen Begleitmaterials auf Dauer an Dritte veräußern oder verschenken, vorausgesetzt, der erwerbende Dritte erklärt sich mit der Weitergeltung der vorliegenden Vertragsbedingungen dem Anwender gegenüber einverstanden.
(2) Der Anwender muß die vorliegenden Vertragsbedingungen sorgfältig aufbewahren. Vor der Weitergabe der Software muß er sie dem neuen Anwender zur Kenntnisnahme vorlegen. Sollte der Anwender zum Zeitpunkt der Weitergabe die vorliegenden Vertragsbedingungen nicht mehr im Besitz haben, ist er verpflichtet, ein Ersatzexemplar beim Hersteller anzufordern. Die entstehenden Versandkosten trägt der Anwender.
(3) Im Falle der Weitergabe muß der Anwender dem neuen Anwender sämtliche Programmkopien einschließlich gegebenenfalls vorhandener Sicherheitskopien übergeben oder die nicht übergebenen Kopien vernichten. Infolge der Weitergabe erlischt das Recht des alten Anwenders zur Programmnutzung.
Der Anwender ist verpflichtet, der Informationspflicht nach 11 dieses Vertrages nachzukommen.

5 Rekompilierung und Programmänderungen

(1) Die Rückübersetzung des überlassenen Programmcodes in andere Codeformen (Rekompilierung) sowie sonstige Arten der Rückerschließung der verschiedenen Herstellungsstufen der Software (Reverse-Engineering) einschließlich einer Programmänderung sind im privaten Bereich zulässig.
(2) Die Entfernung eines Kopierschutzes oder ähnlicher Schutzroutinen ist nur zulässig, sofern durch diesen Schutzmechanismus die störungsfreie Pogrammnutzung beeinträchtigt oder verhindert wurde. Für die Beeinträchtigung oder Verhinderung störungsfreier Benutzbarkeit durch den Schutzmechanismus trägt der Anwender die Beweislast.
(3) Die in den voranstehenden beiden Absätzen angesprochenen Handlungen dürfen nur dann kommerziell arbeitenden Dritten überlassen werden, die mit dem Lieferanten in einem potentiellen Wettbewerbsverhältnis stehen, wenn der Lieferant die gewünschten Programmänderungen nicht gegen ein angemessenes Entgelt vornehmen will. Der Anwender muß dem Lieferanten eine hinreichende Frist zur Prüfung der Auftragsübernahme einräumen.
(4) Urherberrechtsvermerke, Seriennummern sowie sonstige der Programmidentifikation dienende Merkmale dürfen auf keinen Fall entfernt oder verändert werden.

6 Gewährleistung

(1) Mängel der gelieferten Software einschließlich der Handbücher und sonstiger Unterlagen werden vom Lieferanten innerhalb der Gewährleistungsfrist von sechs Monaten ab Lieferung nach entsprechender Mitteilung durch den Anwender behoben. Dies geschieht nach Wahl des Lieferanten durch kostenfreie Nachbesserung oder durch Ersatzlieferung in Form eines Updates.

(2) Kann der Mangel nicht innerhalb angemessener Frist behoben werden, oder ist die Nachbesserung oder Ersatzlieferung aus sonstigen Gründen als fehlgeschlagen anzusehen, kann der Anwender nach seiner Wahl Herabsetzung der Vergütung (Minderung) oder Rückgängigmachung des Vertrages (Wandelung) verlangen. Von einem Fehlschlagen der Nachbesserung oder der Ersatzlieferung ist erst auszugehen: wenn dem Lieferanten hinreichende Gelegenheit zur Nachbesserung oder Ersatzlieferung eingeräumt wurde, ohne daß der gewünschte Erfolg erzielt wurde; wenn die Nachbesserung oder Ersatzlieferung unmöglich ist; wenn sie vom Lieferanten verweigert oder unzumutbar verzögert wird; wenn begründete Zweifel hinsichtlich der Erfolgsaussichten bestehen; oder wenn eine Unzumutbarkeit aus sonstigen Gründen vorliegt.
(3) Der Anwender weiß, daß nach dem heutigen Stand der Technik die Erstellung völlig fehlerfreier Software nicht möglich ist.

7 Untersuchungs- und Rügepflicht

(1) Der Anwender ist verpflichtet, die gelieferte Software auf offensichtliche Mängel, die einem durchschnittlichen Kunden ohne weiteres auffallen, zu untersuchen. Offensichtliche Mängel, insbesondere das Fehlen von Datenträgern oder Handbüchern sowie erhebliche, leicht sichtbare Beschädigungen des Datenträgers, sind beim Lieferanten innerhalb von zwei Wochen nach Lieferung schriftlich zu rügen. Die Mängel, insbesondere die aufgetretenen Symptome, sind nach Kräften detailliert zu beschreiben.
(2) Mängel, die nicht offensichtlich sind, müssen beim Lieferanten innerhalb von zwei Wochen nach dem Erkennen durch den Anwender gerügt werden.
(3) Bei Verletzung der Untersuchungs- und Rügepflicht gilt die Software in Ansehung des betreffenden Mangels als genehmigt.

8 Haftung

(1) Für Schäden wegen Rechtsmängeln und Fehlens zugesicherter Eigenschaften haftet der Lieferant unbeschränkt. Die Haftung für anfängliches Unvermögen, Verzug und Unmöglichkeit wird auf das Fünffache des Überlassungsentgelts sowie auf solche Schäden begrenzt, mit deren Entstehung im Rahmen einer allgemeinen Software-Überlassung typischerweise gerechnet werden muß. Im übrigen haftet der Lieferant nur für Vorsatz und grobe Fahrlässigkeit auch seiner gesetzlichen Vertreter und Erfüllungsgehilfen, sofern nicht eine Pflicht verletzt wird, deren Einhaltung für die Erreichung des Vertragszwecks von besonderer Bedeutung ist (Kardinalpflicht).
Bei Verletzung der Kardinalpflicht ist die Haftungsbeschränkung für anfängliches Unvermögen entsprechend heranzuziehen.
(2) Der Lieferant haftet nicht für Schäden, die durch Fehlbenutzung der Rechenanlage oder durch mangelnde, regelmäßige Absicherung der Daten in Form von Sicherungskopien entstanden sind.

9 Eigentumsvorbehalt

(1) Der Lieferant behält sich das Eigentum an der dem Anwender gelieferten Software bis zur vollständigen Bezahlung sämtlicher zum Zeitpunkt der Lieferung bestehender oder später entstehender Forderungen aus diesem Vertragsverhältnis vor; bei Bezahlung durch Scheck oder Wechsel bis zu deren Einlösung.
(2) Bei verschuldeten Zahlungsrückständen des Anwenders gilt die Geltendmachung des Eigentumsvorbehalts durch den Lieferanten nicht als Rücktritt vom Vertrag, es sei denn, der Lieferant teilt dies dem Anwender ausdrücklich mit.

10 Transportschäden

Der Anwender ist verpflichtet, eventuelle Transportschäden unverzüglich und schriftlich dem Transporteur zu melden und dem Lieferanten eine Kopie des Schriftverkehrs zuzuschicken, denn alle Sendungen sind über den Lieferanten versichert.

11 Informationspflichten

Der Anwender ist im Falle der Weiterveräußerung der Software verpflichtet, dem Lieferanten den Namen und die vollständige Anschrift des Käufers schriftlich mitzuteilen. Die Adresse des Lieferanten ist der vorderen Umschlagseite der Benutzungsanleitung zu entnehmen.

12 Sonstiges

(1) In diesem Vertrag sind sämtliche Rechte und Pflichten der Vertragsparteien geregelt. Sonstige Vereinbarungen bestehen nicht. Änderungen sind nur in Schriftform und bei Bezugnahme auf diesen Vertrag wirksam und beiderseitig zu unterzeichnen.
(2) Der Gerichtsstand für alle Streitigkeiten aus diesem Vertrag ist, soweit vereinbar, das zuständige Gericht in Frankfurt am Main.
(3) Sollten einzelne Bestimmungen dieser Bedingungen nicht rechtswirksam sein oder ihre Rechtswirksamkeit durch einen späteren Umstand verlieren, oder sollte sich in diesen Bedingungen eine Lücke herausstellen, so wird hierdurch die Rechtswirksamkeit der übrigen Bestimmungen nicht berührt. Anstelle der unwirksamen Vertragsbestimmungen oder zur Ausfüllung der Lücke soll eine angemessene Regelung gelten, die, soweit rechtlich möglich, dem am nächsten kommt, was die Vertragsparteien gewollt haben, soweit sie von der Unwirksamkeit der Bestimmung Kenntnis gehabt hätten.

Inhalt

1. Vorwort zur Version 3 17
2. Die integrierte Entwicklungsumgebung 22
3. Der Spezialist für viel Tipparbeit 42
4. Syntax 85
5. Die Datentypen von MaxonPASCAL 113
6. Variablen und Konstanten 145
7. Prozeduren und Funktionen 147
8. Compiler-Anweisungen und Includefiles 175
9. Units und Module 191
10. MAKE - die integrierte Projektverwaltung 219
11. Standard-Units 235
Anhang 249
Index 307

Inhaltsverzeichnis

1. Vorwort zur Version 3

1.2.1 Disketten 18
1.2.2 Installation 18
1.2.3 Die Komponenten des MaxonPASCAL Systems 18
1.3 Erste Schritte mit MaxonPASCAL 20

2. Die integrierte Entwicklungsumgebung

2.1 Programmstart und Optionen 22
2.2 Compiler und Editor 23
2.3 Das Konzept 24
2.4 Compiler-Funktionen 25
2.4.1 Programme übersetzen 25
2.4.1.1 Das Compiler-Fenster 25
2.4.1.2 Abbruch und Ausstieg 27
2.4.1.3 Auswahl von Fehlermeldungen 27
2.4.1.4 Vom Compiler in den Editor 27
2.4.1.5 „Weiter" 28
2.4.1.6 Programmstart aus dem Compiler-Fenster 28
2.4.1.7 Erneutes öffnen des Compilerfensters 28
2.4.2 Ausführen und Abbrechen 29
2.4.3 Exe- und Objektdateien 29
2.4.4 Daten und Informationen 30
2.5 Features und Optionen 30
2.5.1 Ein kurzer Überblick 30
2.5.2 Der Compiler-Requester 30
2.5.2.1 Die Hauptdatei 30
2.5.2.2 Pfade für Include-Dateien und Units 31
2.5.2.3 Eine „Linker"-Einstellung 31
2.5.2.4 Units im Make-Modus 31
2.5.2.5 Arbeitsspeicher 32
2.5.2.6 Diverse Compiler-Schalter 32
2.5.2.7 Automatisches Erzeugen von Dateien 33
2.5.3 MaxonPASCAL-Systemeinstellungen 33
2.5.3.1 Erzeugen von Icons 33
2.5.3.2 Abhängigkeiten in Make-Dateien 33

2.5.3.3 Protokollieren von Ein- und Ausgaben 34
2.5.3.4 Argumente aus der imaginären Kommandozeile 34
2.5.3.5 Einbindung des Maxon Assemblers 34
2.5.3.6 Debugger 34
2.5.3.7 Stackgröße 35
2.5.3.8 Autosave 35
2.5.3.9 Dateiauswahlbox 35
2.5.4 Abspeichern der Konfigurationsdatei 35
2.5.4.1 Allgemeines 35
2.5.4.2 Der Editor 36
2.5.4.3 Fensterdaten 36
2.5.4.4 Switches und Schalter 36
2.5.4.5 Stringoptionen 36
2.5.4.6 Compiler-Einstellungen 37
2.6 Die ARexx-Schnittstelle 37
2.6.1 Grundlagen 37
2.6.2 Der ARexx-Port 38
2.6.3 Die ARexx-Befehle 39
2.6.3.1 Schließen, öffnen und Beenden 39
2.6.3.2 Übersetzen, Starten und Ähnliches 39
2.6.3.3 Requester und Informationen 39
2.6.3.4 Make 39
2.6.3.5 Schreiben von Dateien 40
2.6.3.6 Optionen speichern und setzen 40
2.6.3.7 Informationen über die Entwicklungsumgebung 40
2.6.3.8 Fehlermeldungen und Fehlerstellenansprung 41

3. Der Spezialist für viel Tipparbeit

3.1 Installation von Edward auf der Festplatte 42
3.2 Aufruf von Edward 42
3.2.1 Start von der Workbench 42
3.1.2 Start aus dem CLI 43
3.2 Eingabe und Bearbeitung von Texten 43
3.2.1 Die Status-Anzeige 43
3.2.2 BACKSP3ACE und DELETE 44
3.2.3 UndoLine-Funktion 44
3.2.4. RETURN, ENTER und TAB 44
3.2.5 Die Cursor-Tasten und der Ziffernblock 45
3.2.6 Der Schieberegler 45

3.2.7 Die Maus 45
3.3 Text 46
3.3.1 Erzeugen von Texten 46
3.3.2 Wahl eines bestimmten Textes 46
3.3.3 Freigeben eines Textes 47
3.3.4 Löschen des aktuellen Textes 47
3.3.5 Drucken eines Textes 47
3.4 Dateioperationen 48
3.4.1 Speichern von Texten 48
3.4.2 Laden von Texten 49
3.5. Blockoperationen 49
3.5.1 Cut, Copy, Paste 49
3.5.2 Spaltenblöcke 51
3.5.3 Laden, Speichern und Drucken 51
3.5.4 Einrücken 52
3.5.5 Umwandeln in Groß- bzw. Kleinbuchstaben 52
3.6 Suchen und Ersetzen 53
3.6.1 Das „Find"-Formular 53
3.6.2. Das „Find & Replace"-Formular 54
3.6.3 Struktur-Check (Klammer-Check) 55
3.6.4 Compiler Error-Datei interpretieren, Fehler anspringen 56
3.6.5 Auf bestimmte Zeilen positionieren 57
3.7 Strukturieren von Texten, Lesezeichen 57
3.7.1Erzeuge Falte 57
3.7.2 Auszeichnen von Zeilen 60
3.7.3 Cursor-Bewegung 60
3.7.4 Lesezeichen (Bookmarks) 60
3.9 „Word-Wrap" 62
3.10 Makros 63
3.10.1 Makros aufzeichnen & abspielen 63
3.10.2 Makros speichern & abspielen 64
3.11 Fenster 65
3.11.1 Erzeugen von Fenstern 65
3.11.2 Wahl eines bestimmten Fensters 66
3.11.3 Schließen von Fenstern 66
3.11.4 Arbeiten mit mehreren Ansichten eines Textes 67

3.12 Verlassen des Editors 68
3.12.1. Unterbrechen der Arbeit 68
3.12.2 Verlassen des Editors 69
3.13 Iconify & HotKey 69
3.13.1 Iconify 69
3.13.2 HotKey 69
4.14 Der Kommandozeilen-Interpreter 70
3.15 Globale Einstellungen 70
3.15.1 Das „Globale Einstellungen"-Formular 70
3.15.2 Das „Seitenformat"-Formular 74
3.16 Benutzerdefinierte Menüs, Tastenbelegung 75
3.16.1 Kurze Einführung in die EBNF-Darstellung 76
3.16.2 Definition eigener Menüs 77
3.16.3 Definition der Tastaturbelegung 78
3.16.4 Definition des HotKeys 80
3.16.5 Einladen einer neuen Definition 81
3.17 ARexx - Kommunikation mit Edward 81
3.18 Zurücknehmen von Text-Änderungen 82

4. Syntax

4.1 Allgemeines 85
4.1.1 Syntaktische Notation 85
4.1.2 Symbole 87
4.1.3 Das Semikolon - das ungeliebte Trennzeichen 88
4.1.4 Kommentare 88
4.1.5 Bezeichner 88
4.2 Programmkopf 89
4.3 Blöcke 89
4.4 Label 90
4.5 Konstanten und Konstantendefinition 90
4.6 Typdefinition 91
4.7 Variablendeklaration 92
4.8 Prozeduren und Funktionen 93
4.8.1 Allgemeines über Prozeduren 93

4.8.2 Parameter 97
4.8.2.1 Wert-Parameter 97
4.8.2.2 Variablenparameter 98
4.8.3 Funktionen 101
4.8.4 Prozedurparameter und andere Spezialitäten 102
4.8.5 Zusammenfassend: Syntax 104
4.9 Anweisungsteil 104
4.9.1 Wertzuweisung 105
4.9.2 Prozeduraufruf 105
4.9.3 Verbundanweisung 106
4.9.4 Fallunterscheidung 106
4.9.5 REPEAT-Schleife 107
4.9.6 WHILE-Schleife 107
4.9.7 FOR-Schleife 107
4.9.8 CASE-Anweisung 108
4.9.9 WITH-Anweisung 110
4.9.10 GOTO-Sprünge 111

5. Die Datentypen von MaxonPASCAL

5.1 Numerische Typen 113
5.1.1 Ganzzahl-Typen 113
5.1.2 Gleitpunktzahlen 117
5.2 Boolean 120
5.3 Char 121
5.4 Aufzählungs- und Ausschnittstypen 121
5.5 Stringtypen 123
5.6 Pointertypen 128
5.7 SET-Typen 129
5.8 Strukturierte Datentypen: Arrays und Records 130
5.9 Typkonvertierungen 136
5.10 Literale für Records und Arrays 139
5.11 Dateitypen 142

6. Variablen und Konstanten

6.1 Standard-Variablen 145
6.2 Vordefinierte Konstanten 146

7. Prozeduren und Funktionen

7.1 Ein- und Ausgabe 147
7.2 Dateien 151
7.3 Arithmetische und andere Funktionen 157
7.4 Pointer 159
7.5 AMIGA-Spezifisches 160
7.6 Nützliche MaxonPASCAL-Spezialitäten 164
7.7 Stringbehandlung 171

8. Compiler-Anweisungen und Includefiles

8.1 Compiler Directives 175
8.1.1 Includefiles 175
8.1.2 Bedingte Compilierung 177
8.1.3 Error 178
8.1.4 Linkersteuerung 179
8.1.5 Compiler-Optionen 179
t - Subrange testen 180
i - Indexbereich bei Array-Zugriff 181
b - Unterbrechen 181
s - Stacküberlauf abfangen 182
a - Überlauf bei arithmetischen Operationen melden 182
8.1.6 Kurzauswertung boolescher Ausdrücke 183
8.1.7 Behandlung von Ein-/Ausgabefehlern 185
8.2 Libraries 186
8.3 Anmerkungen zu den Include-Files 188

9. Units und Module

9.1 Modulare Programmierung: was, warum und wie? 191
9.2 Syntax von Modulen 192
9.3 Objektdateien und Linkersteuerung 194

9.4 Alink-Kompatiblität - Fluch und Segen 196
9.5 Hunks 197
9.6 Dynamische und statische Variablen 198
9.7 Externes und internes Linken mit „PasLib.o“ 199
9.8 Units 201
9.8.1 Ein erstes Beispiel 201
9.8.2 USES 203
9.9 Units - Alles noch 'mal etwas genauer 203
9.9.1 Syntax 203
9.9.2 Units und Module 205
9.9.3 Schachtelung von USES-Aufrufen 206
9.10 Bezeichner-Handhabung in Units 207
9.11 Initialisierung und andere Spezialitäten 209
9.11.1 Initialisierungs-Prozeduren 209
9.11.2 Prüfsummen 210
9.12 Projekte - Modulares Programmieren mit Units 212
9.12.1 Aufräumen im Unit-Verzeichnis 212
9.12.2 Organisation von Projekten 213
9.13 Assembler und PASCAL 215

10. MAKE - die integrierte Projektverwaltung

10.1 Das Prinzip 219
10.1.1 Wozu eine Projektverwaltung? 219
10.1.2 Was ist das überhaupt - ein Projekt? 219
10.1.3 Was hat das ganze mit „Make“ zu tun? 220
10.2 Die Praxis 221
10.2.1 Erstellen von Projekten 221
10.2.2 Hinzufügen von Dateien, umständliche Methode 223
10.2.3 Übersetzen des Projekts 224
10.2.4 Hinzufügen von Dateien, einfache Methode 224
10.2.5 Abhängigkeiten 225
10.2.6 Weitere Funktionen im Make-Fenster 226
10.2.6.1 Projektpflege 226
10.2.6.2 Übersetzung erzwingen 226
10.2.6.3 Dateien in den Editor laden 226

10.2.6.4 Projektdatei wechseln 227
10.2.7 Compileranweisungen für Objektdateien 227
10.2.7.1 Interne Übersetzung 227
10.3 Handgemachte Makedateien 228
10.3.1 Die Grundlagen 228
10.3.2 Der Linkeraufruf 228
10.3.3 Internes Compilieren 229
10.3.4 Externe Übersetzung 230
10.3.5 Weitere Features 230
10.3.5.1 Zeilenverbindung 230
10.3.5.2 Kommentare 231
10.3.5.3 Makros 231
10.3.5.4 Regeln 232

11. Standard-Units

11.1 Crt und PASCALCrt 235
11.2 Printer 242
11.3 Die AMIGA-System-Units 242
11.3.1 Exec1 und Exec 243
11.3.2 GraphTypes und Graphics 243
11.3.3 Intuition 244
11.3.4 Custom 244
11.4 ExecSupport und ExecIO 245

Anhang

A) Compiler-Fehlermeldungen 249
B) Laufzeitfehler 257
D) Die Kommandos des Editors 259
D-1 Eingabe und Bearbeitung von Texten 260
D-2 Verwalten von Texten 264
D-3 Dateioperationen 266
D-4 Blockoperationen 267
D-5 Suchen und Ersetzen 270
D-6 Strukturieren von Texten 272
D-7 Ändern von Optionen für den aktuellen Text 275
D-8 Word-Wrap 276
D-9 Makros 277

D-10 Fenster 278
D-11 Verlassen des Editors 279
D-12 Formulare 280
D-13 Der Kommandozeilen-Interpreter 281
D-14 Ändern von globalen Einstellungen 282
D-15 ARexx - Kommunikation mit Edward 284
D-15.1 Abfrage von Statuswerten Edwards : 284
D-15.2 Laden von Texten aus Edward heraus 287
D-16 UnDo/ReDo - Zurücknehmen von Textänderungen 288
D-17 Verschiedenes 289
E) Editor-Fehlermeldungen. 290
E-1 Informationen 290
E-2 Allgemeine Fehlermeldungen 290
E-3 Fehlermeldungen mit Abfragen 294
E-4 Meldungen mit Abfragen 294
E-5 Fehler bei Betriebssystemoperationen 296
E-6 Eingabefehler 297
E-7 Fehler in den Definitionsdateien 298
E-8 Fehler beim Analysieren von Kommandos 300
E-9 Fehlermeldungen beim Auswerten von Kommandos 301
F) Kurzübersicht 302
1.) Wort-Symbole (reserviert Pascal-Schlüsselwörter): 302
2.) Direktiven: 302
3.) Konstanten 303
4.) Typbezeichner 303
5.) Variablen 304
6.) Prozeduren 304
7.) Funktionen 305
Index 307

1. Vorwort zur Version 3

Seit mehr als fünf Jahren gibt es schon das bekannte und erfolgreiche Amiga-Pascalsystem „KickPascal". Nach einer längeren Pause liegt jetzt wieder eine neue Version vor. Der Generationswechsel drückt sich diesmal nicht nur durch eine neue Versionsnummer, sondern auch durch einen neuen Namen aus - „MaxonPASCAL 3".

Der Compiler ist in seinem Kern größtenteils unverändert geblieben, so daß die Kompatiblität zwischen KickPascal 2 und MaxonPascal 3 auf jeden Fall gewährleistet ist.

Neues Herzstück des Systems ist der komfortable Editor „Edward", der sich schon seit einiger Zeit im MaxonC++-System bewährt hat und übrigens auch separat erworben werden kann. Durch Menüs und Tastenkombinationen - dies kann vollkommen frei konfiguriert und Ihren Bedürfnissen angepaßt werden - wird der Compiler direkt aus dem Editor angesteuert. Einstellungen aller Art werden jetzt über Dialogfenster vorgenommen. Auch sonst arbeitet das System jetzt mit wesentlich mehr Fenstern, was den Komfort und die Übersichtlichkeit steigert.

Die Entwicklung umfangreicher Projekte wird von MaxonPASCAL jetzt besser unterstützt. In die Oberfläche wurde ein leistungsfähiges und doch leicht bedienbares „Make" integriert, das die Übersetzung von Modulen automatisiert. Das Modulkonzept des Compilers wurde zugleich ein wenig erweitert, so daß man jetzt, kurz gesagt, globale Funktionen wie in C oder C++ in „Header-Files" deklarieren kann.

Von den zahlreichen Neuerungen an der Benutzerschnittstelle soll an dieser Stelle nur noch die ARexx-Fähigkeit erwähnt werden. Der Editor besitzt einen ARexx-Port, über den er „ferngesteuert" werden kann, und auf diesen Weg kann natürlich auch der Compiler aufgerufen werden. Sie können den Compiler aber auch separat, d. h. ganz ohne Editor, starten. Der richtet dann einen eigenen ARexx-Port ein und wartet auf dort eingehende Anweisungen. Auf diese Weise können Sie den MaxonPASCAL-Compiler über ARexx-Scripte in beliebige andere Systeme integrieren und so z. B. einen anderen Editor einbinden.

Zu guter letzt soll nicht vergessen werden, daß die Units und Includes wieder einmal an neue Betriebssystemversionen angepaßt wurden. Ab sofort gehören die Systemdateien für Version 3.0 zum Lieferumfang. Selbstverständlich werden wir auch in Zukunft mit der Weiterentwicklung des Amiga-Systems Schritt halten und alle neuen Amiga-Systemincludes für MaxonPASCAL anpassen.

1.2 Ein erster Überblick

1.2.1 Disketten

MaxonPASCAL wird auf mehreren Disketten ohne Kopierschutz geliefert. Weitere aktuelle Angaben finden Sie in der „ReadMe"-Datei auf der ersten Diskette. Diese Datei sollten Sie auf jeden Fall als erstes lesen.

1.2.2 Installation

Um das System auf Ihrer Festplatte zu installieren, sollten Sie sich zuerst ein geeignetes Verzeichnis anlegegen, z. B. „WORK:MaxonPASCAL".

Nun starten Sie einfach das Installationsprogramm von der ersten Diskette. Damit können Sie das System natürlich wahlweise auch auf Disketten installieren.

1.2.3 Die Komponenten des MaxonPASCAL Systems

In diesem Abschnitt sollen die wichtigsten Dateien des MaxonPASCAL Systems kurz erläutert werden. Das soll Ihnen helfen, sich mit den zahlreichen Dateien und Verzeichnissen besser zurechtzufinden.

Bei der Installation wird ein logisches Gerät „PASCAL:" dem Verzeichnis, in das Sie die Installation durchgeführt haben, zugewiesen. Darin befindet sich ein Unterverzeichnis „Bin", in dem die meisten ausführbaren Dateien stehen.

"MPASCAL" ist die integrierte Entwicklungsumgebung. Wenn die Installation korrekt und erfolgreich abgelaufen ist, können Sie einfach einmal versuchen, dieses Programm zu starten.

MPASCAL enthält das komplette Entwicklungssystem einschließlich Compiler und Projektverwaltung - nur der Editor muß noch dazugeladen werden. Während der Installation wird der Editor „Edward" in ein Verzeichnis „Edward:" abgelegt. MPASCAL lädt ihn von dort automatisch nach. Sie können Edward aber auch allein ohne Compiler aufrufen.

Edward benötigt unbedingt eine Datei „Edward:Edward.DEF", in der die Menü- und Tastenbelegung der Entwicklungsumgebung abgespeichert ist. Diese Datei wird bei der Installation automatisch erzeugt. Im Edward-Handbuch finden Sie eine ausführliche Anleitung, wie Sie diese Datei ändern können, um das MaxonPASCAL-System nach Ihren persönlichen Vorstellungen und Wünschen zu konfigurieren.

Sowohl MPASCAL als auch Edward benötigen die „rct.library". Diese sollte während der Installation in Ihr „Libs:"-Verzeichnis kopiert worden sein.

Neben einigen nützlichen anderen Funktionen erleichtert die „rct.library" insbesondere die Programmierung von Benutzerschnittstellen, die mit dem Oberflächengenerator „MakeAPP" erzeugt wurden. Ein solches Programm benötigt dann aber nicht nur diese Library, sondern auch eine

„.RCT"-Datei, in der die eigentliche Oberfläche abgespeichert ist. Für MPASCAL heißt diese Datei „MPASCAL.RCT" und wird im Verzeichnis „PASCAL:Bin" erwartet, während die Edward-Oberfläche in „EdwardRes.RCT" im Verzeichnis „Edward:" abgespeichert ist.

Um MPASCAL zu starten, müssen also folgende Dateien installiert sein und im richtigen Verzeichnis liegen:

- PASCAL:Bin/MPASCAL
- PASCAL:Bin/MPASCAL.RCT
- EDWARD:Edward
- EDWARD:Edward.DEF
- EDWARD:MPASCAL3.DEF
- EDWARD:EdwardRes.RCT
- LIBS:rct.library

In der Entwicklungsumgebung können Sie über Requester und dergleichen eine ganze Menge Einstellungen vornehmen. Damit Sie das nicht bei jedem Start des Systems wiederholen müssen, können Sie die Einstellungen natürlich auch abspeichern, und zwar in zwei Dateien im Verzeichnis „S:":

- „S:Edward.CONF" für Editor-Einstellungen,
- „S:MPASCAL.config" für Compiler-Optionen.

Auch diese beiden Dateien werden bei der Installation automatisch angelegt. Wenn Sie fehlen, macht das aber nichts, da sowohl MPASCAL als auch Edward sinnvolle Default-Werte besitzen.

Vorzugsweise sucht das MaxonPASCAL-System ohnehin nicht nach den Dateien im „S:"-Verzeichnis, sondern versucht nach Möglichkeit, aus dem aktuellen Verzeichnis folgende Dateien zu lesen:

- „Edward.SESSION" für Editor-Einstellungen,
- „MPASCAL.SESSION" für Compiler-Optionen.

Diese beiden Dateien enthalten nicht nur sämtliche Einstellungen und Optionen, sondern auch Informationen darüber, welche Texte geladen und welche Fenster an welchen Positionen geöffnet sind. Die „Session"-Dateien werden aber nicht bei der Installation erzeugt, sondern auf Verlangen von MPACAL bzw. Edward selbst ins aktuelle Verzeichnis geschrieben.

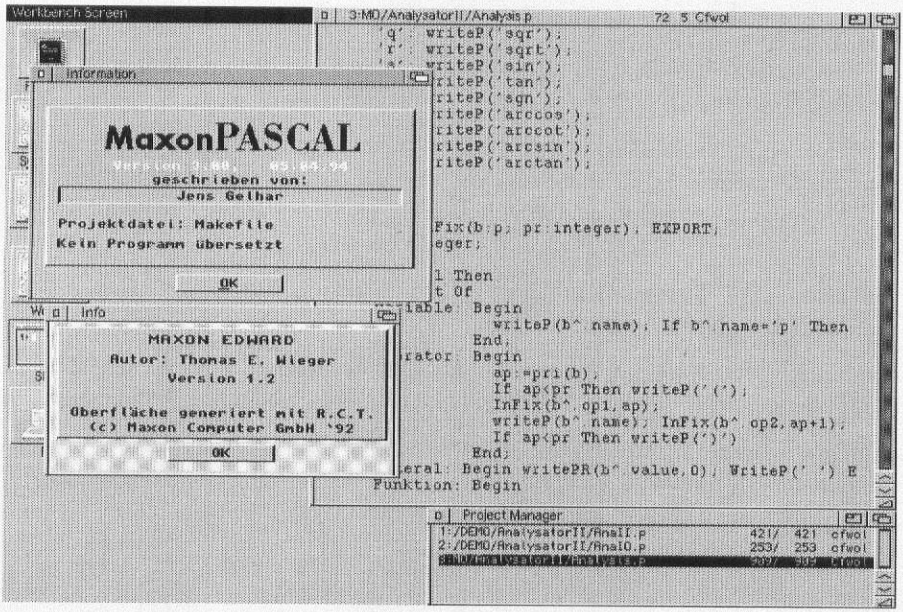
Ferner sollte nach erfolgreicher Installation auch irgendwo ein Verzeichnis namens „HOTHELP:" existieren, in dem alle Dateien und Verzeichnisse liegen, die zum Online-Hilfesystem „HotHelp" gehören.

Um sinnvoll mit MaxonPASCAL arbeiten zu können, benötigen Sie auch Units und Include-Dateien. Die Standard-Units befinden sich normalerweise im Verzeichnis „PASCAL:UNITS", während in „PASCAL:Include" sowohl die Standard-Includes von MaxonPASCAL als auch die jeweils aktuellsten Amiga-OS-Includes von Commodore liegen. Wenn Sie diese Pfadnamen ändern wollen, sollten Sie wissen, was sie tun, und dieses Benutzerhandbuch aufmerksam gelesen haben.

1.3 Erste Schritte mit MaxonPASCAL

Die integrierte Entwicklungsumgebung „MPASCAL" kann wahlweise von der Workbench oder vom CLI aus gestartet werden. Dabei besitzt das Programm keine eigene Benutzeroberfläche, sondern lädt den Editor „Edward" nach. Folglich muß dieser Editor korrekt installiert sein. Wird der Editor nicht gefunden, startet MaxonPASCAL nicht und gibt eine entsprechende Nachricht aus.

Nach dem Start landen Sie automatisch im Editor. Hier können Sie Ihre PASCAL-Programme schreiben und bearbeiten. Dem Editor „Edward" ist ein separates Handbuch gewidmet, deshalb soll hier nicht näher auf die Editor-Funktionen eingegangen werden.



Am besten machen Sie sich jetzt ein wenig mit dem Editor vertraut, indem Sie ein kleines Programm schreiben. Sie könnten z. B. folgendes unvermeidliche und traditionsreiche Prográmmchen eingeben:

```
PROGRAM Hello;
BEGIN
  writeln('Hello World!')
END.
```

Nun haben Sie ein Programm als Quelltext vorliegen und können den Compiler ausprobieren. Zuerst sollten Sie den Text abspeichern, um ihm so einen sinnvollen Namen zu geben, mit dem der Compiler auch etwas anfangen kann. Dann wählen Sie einfach den Menüpunkt **„übersetzen"** aus der Menüleiste und warten ab, was geschieht.

Es öffnet sich ein Fenster, in dem MaxonPASCAL gegebenenfalls Fehlermeldungen oder Warnungen ausgibt. Tritt ein Fehler auf, wird er in diesem Fenster gemeldet und der entsprechende Quelltextbereich angezeigt. Sie können dann eine Fehlermeldung anwählen und mit den Gadgets ***„Anspringen"*** und ***„In Editor"*** direkt die entsprechende Textstelle im Editor anspringen.

Wenn das System korrekt installiert wurde und ihr Programm ebenfalls fehlerlos ist, sollte das Fenster aber ebenso schnell verschwinden, wie es aufgetaucht ist. Nun können Sie die Menü-Funktion **„Ausführen"** anwählen und das Programm starten. Auch dabei öffnet sich ein Fenster, in dem (wer hätte das gedacht!) „Hello, World!" ausgegeben wird. Außerdem werden Sie freundlich aufgefordert, die <Return>-Taste zu drücken, denn sinnvollerweise schließt MPASCAL das Fenster nicht, bevor Sie Gelegenheit hatten, die Ausgabe zu lesen.

Sie können das Programm natürlich noch einmal starten. Falls Sie seit der letzten Übersetzung etwas am Quelltext verändert haben, wird der Compiler automatisch neu gestartet.

2. Die integrierte Entwicklungsumgebung

2.1 Programmstart und Optionen

Wenn Sie MPASCAL vom CLI bzw. aus einer Shell starten, können Sie optional einige Parameter angeben. Als erstes können Sie einen Dateinamen angeben, z. B.

```
run MPASCAL Demos:prg/hello.p
```

Sofern die angegebene Datei existiert und lesbar ist, wird sie in den Editor geladen. Da der Edward mehrere Texte verwalten kann, können Sie natürlich auch mehrere Dateinamen angeben.

Wenn Sie eine andere Konfigurationsdatei als „S:MPACAL.config" benutzen wollen, können sie ein zusätzliches Argument mit vorangestelltem „-c" angeben.
Ein Beispiel: `run MPASCAL test.c -c PASCAL:spezial.config`

Zwischen „-c" und dem Dateinamen darf auch ein Leerzeichen stehen.

MaxonPASCAL besitzt eine integrierte Projektverwaltung, allgemein bekannt als „Make". Sie können gleich beim Programmstart eine solche Make-Datei öffnen, und zwar mit der Option „-m":

```
run MPASCAL -m plot.makefile.
```

Natürlich lassen sich diese Optionen und Argumente in beliebiger Reihenfolge kombinieren, z. B.

```
MPASCAL fclass.p -m wagga.makefile -c df1:s/MPASCAL.config.
```

Seit Version 3.0 besitzt MaxonPASCAL einen eigenen ARexx-Port. Wenn Sie als einziges Argument „-r" angeben, also z. B. run MPASCAL -r startet MaxonPASCAL im ARexx-Modus. Dabei wird Edward nicht geladen. Statt dessen wird ein ARexx-Port namens „MAXONPASCAL" eingerichtet, mit dem das System über ein ARexx-Script gesteuert werden kann. Auch für das Nachladen des Editors und die Kommunikation zwischen den beiden Programmen müssen Sie dann (über geeignete ARexx-Programme) selbst sorgen. So können Sie einen beliebigen - oder auch überhaupt keinen - Editor in die Umgebung einbinden.

Hinter der Option „-r" kann optional noch ein Portname angegeben werden, z. B. „run MPASCAL -r Wagga".

In diesem Fall wird der ARexx-Port nicht „MAXONPASCAL", sondern „WAGGA" genannt. Man beachte, daß der Portname in Großbuchstaben umgewandelt wird. In jedem Fall aber gilt, daß MaxonPASCAL für eindeutige Portnamen sorgt. Falls z. B. schon ein Port „MAXONPASCAL" existiert, werden so lange „MAXONPASCAL1", „MAXONPASCAL2", MAXONPASCAL3" usw. ausprobiert, bis ein Name gefunden wird, den es im ganzen System noch nicht gibt.

Alles weitere über den ARexx-Port und was Sie damit anfangen können entnehmen Sie bitte Abschnitt x.y.

2.2 Compiler und Editor - eine harmonische Beziehungskiste

Wie oben schon angedeutet, ist der Editor „Edward" ein eigenständiges Programm, das auch ohne den Compiler benutzt werden kann. MaxonPASCAL lädt ihn einfach nach und startet ihn als zweiten Prozess, läßt aber keinen Zweifel daran aufkommen, daß er der Boß ist.

Weniger blumig ausgedrückt, richtet MaxonPASCAL einen Port ein, über den sich Editor und Compiler Nachrichten zuschicken. Wird im Edward vom Benutzer eine Compiler-Funktion aufgerufen, etwa durch eines der Pull-Down-Menüs oder eine entsprechend belegte Tastenkombination, läßt Edward dem Compiler eine entsprechende Nachricht zukommen.

Ich erzähle ihnen diese ganzen implementatorischen Details, damit Sie verstehen können, wie Sie sich das MaxonPASCAL-System selbst konfigurieren können. Der Schlüssel dazu ist die „.DEF"-Datei des Edward. Hier werden alle Menüs und Tastenbelegungen des Editors abgelegt, und hier muß irgendwie die Benutzerschnittstelle des Compilers integriert werden, auch wenn der Editor nicht die geringste Ahnug hat, was ein Compiler ist und wozu er gut ist.

Deshalb hat der Edward eine Spezial-Funktion namens *„SendAppComm"*, die als Argument wiederum eine Compiler-Funktion erwartet. Beispielsweise gibt es eine MaxonPASCAL-Anweisung namens *„compile"*. Wird nun vom Benutzer, also Ihnen, im Edward ein Menü ausgewählt, das mit der Funktion *'SendAppComm „compile"'* belegt ist, schickt Edward das *„compile"* als Nachricht an MPASCAL und kümmert sich dann nicht weiter drum. MaxonPASCAL dagegen erhält diese Anweisung, fordert vom Edward noch ein paar zusätzliche Daten an und beginnt dann freudig mit der Übersetzung. Natürlich kann man nicht nur ein Menü, sondern auch eine Taste oder Tastenkombination mit diesen *„SendAppComm"*-Anweisungen belegen.

Auf diese Weise sind der Compiler und der Editor vollkommen voneinander abgekoppelt, und Edward braucht nichts über MPASCAL zu wissen. Nun kommt es natürlich vor, daß Sie den Edward einmal „einfach so" ohne Compiler benutzen wollen. Deshalb läßt Edward einen kleinen „Preprozessor" auf das „.DEF"-File los und filtert je nach Modus Teile aus dem Text aus - ähnlich wie ein C-Preprozessor bei „#ifdef"-Zeilen. Auch dieses Feature ist in ihrem Edward-Handbuch beschrieben. MPASCAL teilt dem Edward mit, daß ein Symbol namens „MPASCAL3" definiert sein soll. Nun können Sie Dinge wie „#ifdef MPASCAL3" in Ihre DEF-Datei schreiben, um Definitionen je nach Modus ein- oder auszublenden. Insbesondere werden Sie sich um sämtliche Menü-Definitionen des Compilers eine „#ifdef MPASCAL3 ... #endif"-Klammer legen wollen, wie es auch in der mitgelieferten Standard-Datei der Fall ist.

Langer Rede kurzer Sinn: Wenn im folgenden von irgendwelchen Menüs die Rede ist, werden dabei jedesmal Hinweise wie „...oder wie auch immer Sie diese Funktion konfiguriert haben" unterschlage. Gemeint ist damit jeweils immer die Funktion, die in der Standard-Konfiguration auf dem entsprechenden Menü liegt.

Es gibt übrigens standardmäßig folgende Menüs, mit den nachstehend angegebenen Compiler-Funktionsnamen:

*** Compiler**

- Übersetzen compile
- Compile Datei compfile
- Ausführen run
- Abbrechen break
- Exe-Datei exefile
- Objektdatei objectfile
- Information info
- Fehlermeldungen errors

*** Make**

- Projektverwaltung make
- Neues Makefile newmk
- Makefile laden loadmk
- Text aus Editor aufnehmen addedittx
- Text und Objekt aufnehmen addtxobj
- Make verlassen leavemk

*** Optionen**

- System opsys
- Compileropcomp
- Optionen sichern saveconf

2.3 Das Konzept

Auch auf die Gefahr hin, daß ich Sie damit langweilen könnte, möchte ich trotzdem noch ein paar grundsätzliche Anmerkungen loswerden.

Prinzipiell kann MaxonPASCAL in zwei verschiedenen Modi arbeiten: Im „normalen" Modus, in dem man vor allen kleinere Programme (mit nur einem Modul) schreibt, und im „Make"-Modus. Da der Projekverwaltung „Make" ein eigenes Kapitel gewidnet ist, befassen wir uns zunächst nur mit dem Standard-Modus, der natürlich auch aktiv ist, wenn Sie das MaxonPASCAL-System starten.

Auch wenn Edward beliebig viele Texte gleichzeitig bearbeiten kann, verwaltet MaxonPASCAL immer nur ein Programm. Das hat zum einen Gründe, die mit dem Speicherplatz zusammenhängen, und resultiert zum anderen aus der Beobachtung, daß das in der Praxis nicht besonders relevant ist, weil man normalerweise zwar durchaus mit mehreren Texten, aber an nur einem Programm arbeitet.
Die Funktion „compile" übersetzt stets den Text im gerade aktiven Fenster. Bei „run" wird immer zuerst folgendes geprüft:

- **Liegt überhaupt ein übersetztes Programm vor?**
- **Befindet sich der Quelltext dieses Programms im aktiven Editor-Fenster?**
- **Blieb er seit der letzten Übersetzung unverändert?**

Falls eine der drei Fragen mit „Nein" beantwortet werden muß, wird die „*compile*"-Funktion aufgerufen und damit der Text im aktiven Fenster übersetzt.

So, das sollte gesagt werden, damit Sie beim Rumprobieren mit MaxonPASCAL (was Sie wahrscheinlich schon längst getan haben) nicht in vermeidbare Verwirrung stürzen (was dabei hoffentlich nicht geschehen ist). Und nun folgt endlich eine ausführliche Beschreibung der Funktionen von MaxonPASCAL:

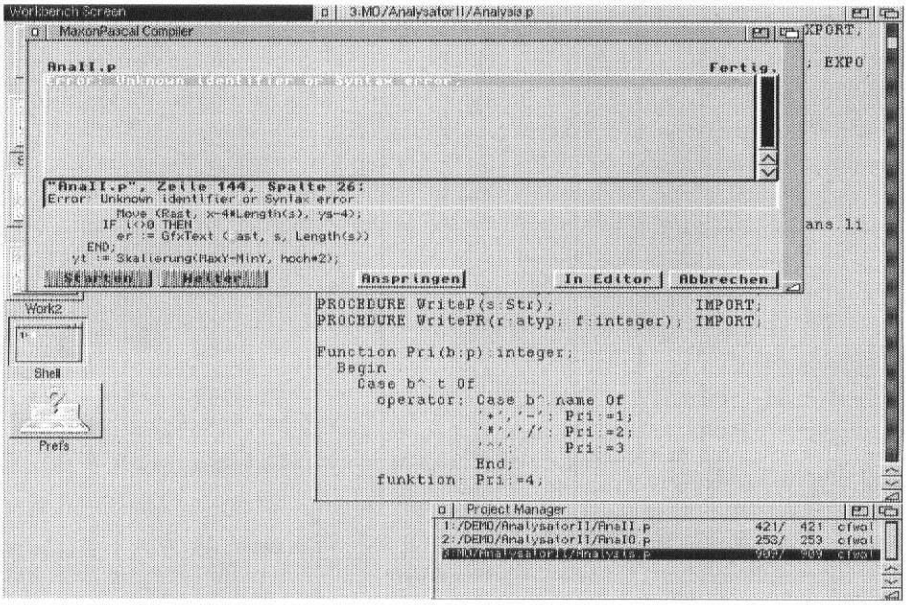
2.4 Compiler-Funktionen

2.4.1 Programme übersetzen

2.4.1.1 Das Compiler-Fenster

Am besten fangen wir mit der Funktion an, wegen der Sie MaxonPASCAL überhaupt gekauft haben, nämlich dem Übersetzen von Programmen.

Wenn Sie diese Funktion auswählen, wird - je nach Modus - entweder das Programm im gerade aktiven Editor-Fenster übersetzt, oder das integrierte „Make" entscheidet selbst, welche Objektdateien neu übersetzt werden sollen. In jedem Fall öffnet sich aber ein Fenster, das in fünf Bereiche aufgeteilt ist.



Am oberen Rand befindet sich die Statuszeile. Hier wird stets links der Name der gerade übersetzten Datei angezeigt, während MaxonPASCAL rechts sagt, was gerade getan wird bzw. zuletzt getan wurde.

Am besten probieren Sie es einmal aus: laden Sie einen Pascal-Quelltext (z. B. eines der Beispielprogramme) in den Editor und wählen Sie den Menüpunkt **„Übersetzen"** aus. Wenn alles klappt, wird oben rechts in schneller Folge (bei kurzen Programmen wohl zu schnell zum Mitlesen) das folgende Ausgegeben:

- **„Compiling":**

  Der Compiler liest und analysiert das Programm.

- **„Linking":**

  Der Linker bindet das erzeugte Programm mit den nötigen Bibliotheksfunktionen zusammen.

- **„Fertig":**

  Das Programm ist übersetzt und ausführbar. Wurden keine Fehler oder Warnungen ausgegeben, verschwindet das Fenster bei dieser Meldung automatisch.

Natürlich kann dabei beliebig viel schief gehen, beispielsweise könnte das Programm fehlerhaft sein. Fehlermeldungen und Warnungen aller Art werden in dem Rahmen unterhalb der Statuszeile ausgegeben. MaxonPASCAL gibt immer höchstens so viele Fehler aus, wie in diesen Rahmen passen (das sind zehn Stück) und wartet auf eine Reaktion des Benutzers, falls die Höchstzahl erreicht wurde.

Eine solche einzeilige Fehlermeldung allein ist meist natürlich nicht aussagekräftig genug. Deshalb kann sich der Programmierer jeweils zu einer Meldung ausführlichere Informationen anzeigen lassen:

Unterhalb des Fehlermeldungs-Rahmen befindet sich ein weiterer Rahmen, in den normalerweise zwei Zeilen ausgegeben werden. Darin stehen Quelltextdatei, Zeilen- und Spaltennummer des gerade angewählten Fehlers und noch einmal eine Wiederholung der schon im oberen Rahmen ausgegebenen Fehlermeldung. Unterhalb dieser Anzeige sehen Sie einen Ausschnitt aus dem Quelltext mit einem Pseudo-Cursor an der fraglichen Position.Wenn allerdings der Linker Fehler meldet, steht anstelle der Sourceposition (was weiß ein Linker schon von Quelltexten!) schlicht **„Linker Error"**, und dort, wo sonst der Quelltext-Ausschnitt dargestellt wird, steht eine eventuell verkürzte Liste der beim Zusammenbinden aufgetretenen Fehler, also hauptsächlich Namen von undefinierten oder gleich mehrfach definierten Linker-Symbolen.

Weniger häufig treten beim Übersetzen unter dem Titel **„Fatal Error"** Fehler bei der Codegenerierung oder beim Assemblieren auf. Meist sind diese auf Speichermangel zurückzuführen, und dementsprechend heißen sie **„Out of Memory"**, **„Workspace overflow"** oder wie auch immer.

Last not least gibt es ganz unten im Compiler-Fenster noch eine Gadgetleiste. Zunächst ist hier nur das Gadget ganz rechts mit der Aufschrift ***„Abbrechen"*** anwählbar. Bei Bedarf werden die übrigen Gadgets aktiviert.

2.4.1.2 Abbruch und Ausstieg

Im Gegensatz zu einfachen Kommandozeilen-Compilern besitzt das MaxonPASCAL-Fenster eine Vielzahl von Knöpfen und Funktionen, an denen der Pascal-Programmierer herumspielen kann. Alle Gadget-Funktionen sind, um den Bedürfnissen der „echten Programmierer" entgegenzukommen, auch durch Tasten oder Tastenkombinationen bedienbar.

Beim Start des Compilers können Sie im Compiler-Fenster zunächst nur zwei Dinge tun: Das Fenster schließen oder die Übersetzung abbrechen.

Mit einem Klick auf das Schließ-Gadget wird die Übersetzung abgebrochen und das Fenster geschlossen. Die Taste <ESC> hat die gleiche Wirkung.

Außerdem gibt es unten rechts das bereits erwähnte ***„Abbrechen"***-Gadget. Man benutzt es, wenn man den Compiler zwar abbrechen, aber noch irgendwelche im Window ausgegebenen Meldungen in Ruhe lesen möchte. Die kanonische Tastenkombination <Control>+<C> entspricht diesem Gadget.

Falls der Compiler aber schon fertig ist, wird das Compiler-Fenster bei einem Klick auf ***„Abbrechen"*** geschlossen.

2.4.1.3 Auswahl von Fehlermeldungen

Laut Murphy geht bei so ziemlich jedem Compiler-Lauf etwas schief, und meist sogar etwas mehr als nur „etwas". Deshalb bietet das Compiler-Fenster Platz für bis zu zehn Fehlermeldungen auf einmal.

Eine dieser jeweils einzeiligen Meldungen ist immer invertiert und somit hervorgehoben, und zu genau dieser Meldung werden unterhalb des Rahmens zusätzliche Informationen angezeigt. Mit einem Mausklick auf eine Fehlerzeile macht man diese zur gerade privilegierten, invertierten Meldung.

Das geht aber auch über die Tastatur: Mit der <Leertaste> durchläuft man die Fehlermeldungen von oben nach unten, und mit <Backspace> in die andere Richtung. Alternativ können Sie auch die Cursortasten <Auf> und <Ab> benutzen.

2.4.1.4 Vom Compiler in den Editor

Der besondere Witz an einer integrierten Umgebung ist das perfekte Zusammenspiel von Compiler und Editor, insbesondere das automatische Anspringen von Fehlerstellen.

Das funktioniert bei MaxonPASCAL denkbar einfach: Sobald eine Fehlermeldung im Compiler-Fenster ausgegeben wird, werden auch zwei Gadgets mit der Aufschrift ***„Anspringen"*** bzw. ***„In Editor"*** aktiviert. Nun wählen Sie wie oben beschrieben Ihre Wunsch-Fehlermeldung aus (das geht übrigens auch, während der Compiler noch arbeitet) und klicken ***„Anspringen"*** an. Prompt lädt Edward die zugehörige Quelltextdatei (falls der Text nicht ohnehin schon im Editor vorliegt) und setzt den Cursor an die bemängelte Position. Nun können Sie den Fehler berichtigen - vorausgesetzt natürlich, der Compiler läuft nicht noch und hat ein „Lock" auf diesen Text eingerichtet. Das Compiler-Fenster wird dabei nicht geschlossen, so daß Sie noch weitere Fehlerstellen anspringen können.

Die Integration von Compiler und Editor geht allerdings nicht so weit, daß im Compiler-Fenster Quelltext-Veränderungen beachtet würden. Wenn Sie z. B. zuerst einen Fehler berichtigen und dabei einige Zeilen einfügen oder löschen müssen, merkt MaxonPASCAL nichts davon. Sollten Sie also anschließend ein weiter unten im Quelltext gelegene Fehlerstelle anspringen lassen, landen Sie gnadenlos in der Zeile mit der Nummer, in der der Fehler lag, als er bemerkt wurde - und durch die inzwischen vorgenommenen Änderungen stimmt diese Zeilennummer inzwischen vielleicht nicht mehr, und Sie landen einige Zeilen zu hoch oder tief. Das dürfte aber in der Praxis kaum ein Problem darstellen, denn im Compiler-Fenster wird immer noch der ursprüngliche Quelltext mit dem Cursor an der richtigen Stelle angezeigt, und durch einen kurzen Textvergleich zwischen Compiler- und Editorfenster können Sie schnell die wirkliche Fehlerposition „zu Fuß" ansteuern. Außerdem ändert man in der Praxis bei Compilerfehlern meist sowieso nur die fragliche Zeile, ohne großartiges Löschen und Einfügen von Zeilen.

Das Gadget ***„Anspringen"*** ist nebenbei bemerkt mit dem Shortcut <A> belegt.

Manchmal tritt während der Übersetzung nur ein einziger Fehler auf, der vielleicht noch den einen oder anderen Folge-Fehler nach sich zieht, aber trotzdem nur eine einzige Änderung am Quelltext notwendig macht. Deshalb gibt es auch noch das Gadget ***„In Editor"***. Hier wird im Editor die gerade angewählte Fehlerposition angesprungen (genau wie beim ***„Anspringen"***-Gadget) und gleichzeitig das Compiler-Fenster geschlossen - wobei natürlich auch der Compiler abgebrochen wird, sollte er noch am arbeiten sein.

Alternativ können Sie auch die Tasten <Return> oder <E> drücken, und ebenso gut können sie auch mit der Maus einen Doppelklick auf die Fehlermeldung machen.

2.4.1.5 „Weiter"

Treten beim Übersetzen bzw. Linken mehr Fehler auf, als im Rahmen dargestellt werden können (das sind bekanntlich zehn Stück), wartet der Compiler erst einmal ab, und das ***„Weiter"***-Gadget wird anwählbar. Nach einem Klick auf selbiges läuft der Compiler wieder weiter, bis erneut zehn Fehlermeldungen voll sind. Meist ist es aber nicht unbedingt sinnvoll, nach mehr als zehn Fehlern noch weiter zu übersetzen, denn normalerweise kommt dabei nicht mehr so ganz viel sinnvolles 'raus. Deshalb wird man in der Regel das ***„Weiter"***-Gadget meiden und lieber die Fehler korrigieren.

2.4.1.6 Programmstart aus dem Compiler-Fenster

Wenn MaxonPASCAL bei der Übersetzung keine Fehler, sondern nur Warnungen ausgegeben hat, wird ein lauffähiges Programm erzeugt. Wenn das dann auch noch zusammengebunden (gelinkt) werden konnte, taucht unten rechts das ***„Starten"***-Gadget auf. Mit einem beherzten Klick darauf können Sie das Programm starten.

2.4.1.7 Erneutes öffnen des Compilerfensters

Wenn Sie das Compiler-Fenster geschlossen haben und es doch noch einmal sehen wollen, können Sie es mit dem Menüpunkt ***„Fehlermeldungen"*** noch einmal öffnen. Wie der Name schon andeu-

tet, sind dann alle Fehlermeldungen der letzten Compilierung wieder sichbar, und auch sonst präsentiert sich das Fenster in dem Zustand, in dem es sich beim Schließen befand.

2.4.2 Ausführen und Abbrechen

Die zweite wichtige Funktion aus dem „Compiler"-Menü heißt **„Ausführen"** und tut auch genau das: Ein Pascal-Programm wird gestartet.

Vorher prüft MaxonPASCAL allerdings noch, ob nicht vielleicht erst noch der Compiler konsultiert werden muß, und startet selbigen automatisch.

Anschließend wird ein Text-Fenster für die Ein- und Ausgaben des Programms geöffnet. Falls das Programm tatsächlich etwas ausgegeben hat, erscheint nach erfolgreicher Programmausführung die Aufforderung „Press Return" - Sie sollen also die <Return>-Taste drücken und so bestätigen, daß Sie die Ausgaben des Programms auch gelesen haben.

Während das Programm läuft, sind verschiedene Funktionen des Systems gesperrt, z. B. darf man es (trotz Multitasking) nicht ein zweites mal gleichzeitig starten, oder man kann den Compiler nicht laufen lassen, weil man dem laufenden Programm ja dadurch den Code unter den Füßen wegziehen bzw. überschreiben würde. Das ist sicher sinnvoll und stört wohl nicht weiter - es sei denn, das Programm ist irgendwo hängengeblieben und läßt sich nicht normal beenden.

Gegen „normale" Endlosschleifen hilft die Compileroption **„Unterbrechen"**. Schwieriger wird es dagegen, wenn der Prozeß abgestürzt ist, auf ein Ereignis wartet, das nie eintreten kann oder die erwähnte Breakpoint-Option einfach nicht benutzt wurde. Deshalb (und nur für solche Fälle!) gibt es im Compiler-Menü die Funktion **„Abbrechen"**, die den gestarteten Prozeß schlicht abtötet und dann das Ausgabe-Fenster schließt. Auf den ersten Blick eine praktische Lösung, aber leider gibt es auf dem Amiga (im Gegensatz z.B. zu Unix) keine saubere Möglichkeit, einen Prozeß aus dem System zu entfernen. Insbesondere werden dabei keine Ressourcen (reservierter Speicher, geöffnete Dateien, Screens und Fenster etc.) freigegeben. Sie müssen davon ausgehen, daß sie auf diese Weise freien Speicher verlieren, wenn Sie einen Prozeß auf diese brutale Art abbrechen. Aber das ist im Zweifel immer noch angenehmer, als das MPASCAL-System neu zu starten oder gar den Rechner zu booten.

2.4.3 Exe- und Objektdateien

Ungeachtet allen damit verbundenen Komforts werden Sie ihre Programme wohl nicht immer nur aus der integrierten Umgebung starten wollen. Deshalb gibt es im „Compiler"-Menü zwei Funktionen, die das übersetzte Programm in Dateien schreiben:

- **„Exe-Datei"** erzeugt eine ausführbare Programmdatei - vorausgesetzt natürlich, ein Programm ist korrekt übersetzt und gelinkt worden.
- **„Objekt-Datei"** schreibt zur zuletzt compilierten Übersetzungseinheit eine Objektdatei, aus der man später ein ausführbares Programm erzeugen kann. Im „Make"-Modus ist diese

Funktion nicht unbedingt sinnvoll, denn dort werden diese Objektdateien ja beim Compilieren automatisch erzeugt.

2.4.4 Daten und Informationen

Mit dem Menüpunkt **„Information"** öffnet man ein Fenster, in dem Informationen über den Status von MaxonPASCAL angezeigt und ständig aktualisiert werden. Neben dem üblichen Rhabarber über Autorschaft und Copyright (klicken Sie doch einfach mal auf den Namen des Autors...) können Sie hier bei Bedarf nachlesen, welchen Quelltext Sie zuletzt übersetzt oder welche Make-Datei gerade geladen ist. Liegt ein übersetztes Programm im Speicher vor, wird auch seine Größe angegeben. Dabei handelt es sich genaugenommen um den Platzbedarf des Programms, wenn es in den Speicher geladen wird, und nicht um die Größe der Exe-Datei.

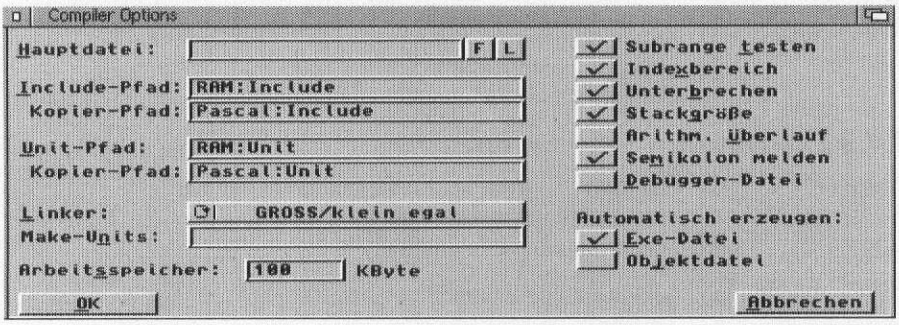
2.5 Features und Optionen

2.5.1 Ein kurzer Überblick

Für viele Programmierer ist das Schönste an einem Compiler, daß man so viele Optionen und Switches hat, an denen man herumspielen kann. Bei MaxonPASCAL nimmt man sämtliche dieser Einstellungen über Requester vor, deren es genau zwei gibt: Einen – **„System"** – für allgemeine Einstellungen, die in irgendeiner Weise mit dem MaxonPASCAL-System zu tun haben und einen für die unvermeidlichen Compiler-Optionen.

Wenn Sie wollen, können Sie beide Requester gleichzeitig öffnen. In jedem der Fenster finden Sie am unteren Rand die obligatorischen Gadgets ***„OK"*** und ***„Abbruch"***, mit denen Sie die vorgenommenen Einstellungen bestätigen bzw. Sie wieder verwerfen können.

2.5.2 Der Compiler-Requester



2.5.2.1 Die Hauptdatei

Wenn Sie eine Unit übersetzt haben, wollen Sie wahrscheinlich anschließend auch testen, ob sie funktioniert. Beim Versuch, sie wie ein Programm zu starten, werden Sie von der MaxonPASCAL-

Umgebung allerdings nur eine entsprechende Fehlermeldung erhalten. Deshalb können Sie eine sogenannte Hauptdatei (in Version 2.x noch „Main File" genannt) angeben, die immer dann automatisch compiliert wird, wenn eine Unit übersetzt wurde. Sinnvollerweise wird dies dann ein Programm sein, das Ihre Unit auch benutzt.

Der Name dieser Datei kann im Textfeld ***„Hauptdatei"*** des Compiler-Requesters eingetippt werden. Gleich rechts von jenem finden Sie ein Gadget, das den Filerequester öffnet und so die Auswahl der Hauptdatei bequemer macht. Ein Klick auf das schlicht mit ***„L"*** beschriftete Symbol löscht den Inhalt des Textgadgets.

Wichtig:

Im Make-Modus wird die Hauptdatei generell ignoriert.

2.5.2.2 Pfade für Include-Dateien und Units

In vier Textfeldern können die Verzeichnisse eingetragen werden, in denen MaxonPASCAL Units und Includes suchen soll. Optional kann jeweils ein zweites Verzeichnis angegeben werden, in das die Dateien automatisch umkopiert werden. Mehr dazu in den Kapiteln über Includes bzw. Units.

2.5.2.3 Eine „Linker"-Einstellung

Der integrierte Linker von MaxonPASCAL ist im wesentlichen zum Amiga-Standard-Linker „ALink" (bzw. der besseren PD-Version „BLink") kompatibel. Jener Linker erkennt Bezeichner nur dann als identisch, wenn Sie in ihrer Groß-/Kleinschreibung übereinstimmen. Beispielsweise sind für ihn „Test" und „test" zwei verschiedene Namen. Dagegen ist man es als Pascal-Programmierer gewohnt, daß die Schreibweise von Bezeichnern keine Rolle spielt. Deshalb haben Sie durch dieses Menü die Wahl, ob der integrierte MaxonPascal-Linker sich Pascal-mäßig verhalten ***(„GROSS/klein egal")*** oder sich an den „ALink"-Standardhalten soll ***(„GROSS/klein beachten")***.

2.5.2.4 Units im Make-Modus

Die neue Projektverwaltung „Make" unterstützt die Verwaltung von Projekten, die in Module gegliedert sind. Dabei wird man aber meist nicht auf Units verzichten wollen. Abschnitt 7.9.2 („Units und Module") verrät Ihnen alles, was Sie bei der Kombination dieser beiden Konzepte zu beachten haben.

Nun gibt es für das „Make" aber noch ein Problem: Da es immer nur die Module recompiliert, bei denen es auch sein muß, kennt es nicht alle im Projekt benutzten Units. Deshalb wird das Nachladen von Units im Make-Modus etwas anders gehandhabt: Sie als Programmierer haben die (sicher nicht übermäßig schwere) Aufgabe, alle im Projekt benutzten Units in das Gadget ***„Make-Units"*** im Compiler-Requester einzutragen. Beim Linken des Projekts werden dann diese (und nur diese) Units dazugeladen. Die Namen der Units sind ohne Punkt und Komma, aber mit Leerzeichen getrennt einzugeben, z. B. „Printer Intuition Windows/PascalCrt".

Hinweis

Außerhalb des Make-Modus hat dieses Gadget keinerlei Bedeutung.

2.5.2.5 Arbeitsspeicher

Als (ehemaliger) KickPascal-Anwender werden Sie sich noch daran erinnern, daß Sie beim Start des KP-Systems immer die Größe des „Workspace", also des Arbeitsspeichers, auswählen mußten. Bei MaxonPASCAL ist das nicht mehr nötig, denn der Editor Edward verwaltet seinen Speicher voll dynamisch.

Nach wie vor braucht aber der Compiler einen zusammenhängenden Speicherblock ausreichender Größe. Diese Form der Speicherverwaltung ist vielleicht unschön, aber zugleich eine der Ursachen für die Geschwindigkeit des MaxonPASCAL-Compilers.

Langer Rede schwacher Sinn: Der Compiler reserviert sich also einen gewissen Arbeistspeicher, und wenn dieser während der Übersetzung überläuft, gibt es eine Fehlermeldung. In diesem Fall sollten Sie den Eintrag ***„Arbeitsspeicher"*** im Compiler-Requester nach Gutdünken hochsetzen. Für mittelgroße Programme ist die Voreinstellung von 80 KByte aber ausreichend.

2.5.2.6 Diverse Compiler-Schalter

Oben rechts im Compiler-Requester finden Sie sieben Schalter, mit denen Sie jeweils bestimmte Compiler-Optionen ein- und ausschalten können:

- Subrange testen:
  Die Grenzen von Ausschnittstypen sind bei Wertzuweisungen zur Laufzeit zu prüfen.
- Indexbereich:
  Bei Array-Zugriffen sind die Grenzen zu testen.
- Unterbrechen:
  An geeigneten Stellen wird im Code eine überprüfung der Signalbits eingebaut, so daß das Programm mit der Tastenkombination Control-C abgebrochen werden kann.
- Stackgröße:
  Ein Überlauf des Laufzeitstacks ist abzufangen.
- Arithmetischer Überlauf:
  Bei Rechenoperationen jeder Art sind Überläufe abzufangen.
- Semikolon melden:
  Fehlende Semikola (also diese „;" Dinger) sind als Fehler zu melden. Das etwas obskure Kickpascal-Feature, fehlende Semikola automatisch ergänzen zu lassen, wird von MaxonPASCAL nicht mehr unterstützt. Allerdings führt ein solcher Fehler aber auch nicht zum Abbruch der Übersetzung.

- Debugger-Datei:
  Ist diese Option aktiviert, werden die Dateien für den demnächst erhältlichen Debugger erzeugt.

2.5.2.7 Automatisches Erzeugen von Dateien

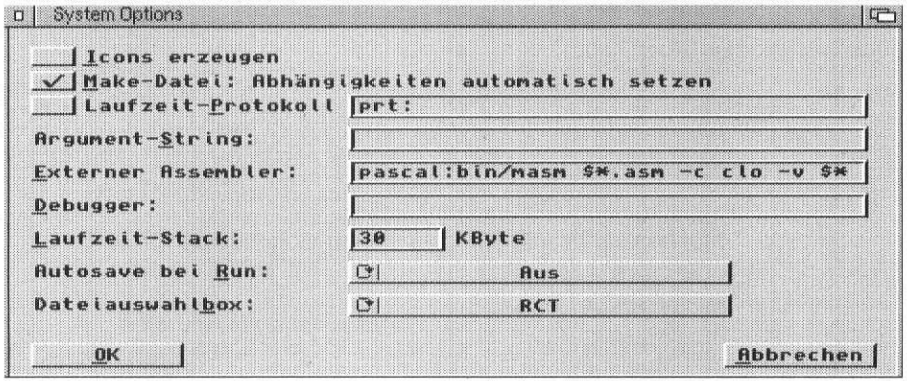
Mit den beiden Schaltern ***„Exe-Datei"*** und ***„Objektdatei"*** können Sie wählen, ob MaxonPASCAL nach jeder Übersetzung das Programm als ausführbare bzw. linkbare Datei schreiben soll. Die Objektdatei-Einstellung hat im Make-Modus keine Wirkung, denn dann werden die Linker-Dateien sowieso geschrieben.

2.5.3 MaxonPASCAL-Systemeinstellungen

Mit dem zweiten Dialogfenster, das Sie mit **„System"** aus dem „Optionen"-Menü öffnen, können diverse Einstellungen der MaxonPASCAL-Umgebung vorgenommen werden.

2.5.3.1 Erzeugen von Icons

"Icons erzeugen" legt fest, ob MaxonPASCAL beim Schreiben von ausführbaren Dateien, Makefiles und Debugger-Dateien jeweils ein Icon erzeugen soll.



2.5.3.2 Abhängigkeiten in Make-Dateien

Die integrierte Projektverwaltung entscheidet souverän, wann welches Modul eines größeren Programms neu compiliert werden muß. Dazu muß sie aber wissen, von welchen Quelltextdateien ein Objekt abhängig ist. Diese Abhängigkeiten kann der Benutzer durchaus von Hand setzen - bequemer und sicherer ist es allerdings, wenn man diese Aufgabe MPASCAL überläßt.

Wenn sie das Symbol ***„Make-Datei: Abhängigkeiten automatisch setzen"*** angewählt haben, paßt MaxonPASCAL beim Compilieren auf, welche Includedateien bei den einzelnen Objektdateien gelesen werden, und trägt diese dann als Abhängigkeit in die Make-Datei ein.

2.5.3.3 Protokollieren von Ein- und Ausgaben

Wenn Sie die Option ***„Laufzeit-Protokoll"*** einschalten, werden sowohl die Ausgaben des Programms als auch die Eingaben des Benutzers in eine Datei geschrieben. Der Dateiname ist im Textgadget einzutragen, wobei „prt:", also der Drucker, als Defaultwert vorgegeben wird.

2.5.3.4 Argumente aus der imaginären Kommandozeile

Beim Start von der Shell kann man Programmen Argumente übergeben, die in PASCAL mit den Variablen *„ParameterStr"* und *„ParameterLen"* (siehe Kap 6.1) ausgewertet werden können. Auch beim Programmstart aus der Entwicklungsumgebung von MaxonPASCAL kann man diese Art der Parameterübergabe simulieren, indem man sie im Textgadget ***„Argument-String"*** eingibt. Diese Argumentzeile wird dann an aus MaxonPASCAL gestartete Programme übergeben.

2.5.3.5 Einbindung des Maxon Assemblers

In MaxonPASCAL beherrscht das integrierte Make auch sogenannte Regeln, die möglichenfalls automatisch angewendet werden, wenn für eine Objektdatei keine Compileranweisung vorhanden ist. Zwei dieser Regeln sind sogar schon „eingebaut": Zum einen die „.p.o"-Regel, die besagt, daß eine auf „.o" endende Datei, zu der eine „.p"-Datei gleichen Namens existiert, als Pascal-Programm compiliert werden kann, und zum anderen die „.asm.o"-Regel für Assemblersourcen. Diese Regel können Sie über das Eingabefeld ***„Externer Assembler"*** frei bestimmen.

Default ist Pascal:Bin/masm $*.asm -c cl -v $*.err

Das „$*" wird (wie immer bei Make-Regeln) durch den endungslosen Dateinamen ersetzt. Also wird für jede auf „.o" endende Objektdatei, für die es im Makefile keine Compileranweisung, dafür aber auf der Platte eine passende „.asm"-Datei gibt, automatisch ein solcher Maxon Assembler-Aufruf erzeugt.

Diese Regel wird auch benutzt, wenn mit dem Menüpunt „Text und Objekt aufnehmen" eine auf „.asm" endende Quelltextdatei in das Makefile aufgenommen wird. Auch in diesem Fall erhalten Sie also automatisch einen passenden Assembleraufruf.

Vielleicht fällt Ihnen die „-v"-Option auf. Der Maxon Assembler ist so weit in das System integriert, daß MaxonPASCAL die MaxonASM-Fehlerdateien interpretieren kann (und genau diese werden durch „-v" erzeugt). Wenn MaxonPASCAL ein Assemblermodul auf diese weise „extern" übersetzen läßt, liest es anschließend diese Fehlerdatei ein und zeigt die Assemblerfehler im normalen Compiler-Fenster an. Prima, nicht?

2.5.3.6 Debugger

Der Menüpunkt **„Debugger"** ruft falls vorhanden die MaxonPASCAL-Version des MaxonSDB auf. Damit MaxonPASCAL auch weiß, was dabei zu tun ist, gibt es das Gadget ***„Debugger"*** im System-Einstellfenster. Hier ist die Shell-Anweisung anzugeben, die von MaxonPASCAL ausgeführt werden soll (bis auf den Dateinamen, denn den hängt MPASCAL einschließlich des Argumentstrings immer automatisch an). Default ist **Pascal:Bin/MaxonSDB <> CON:0/0/640/120/MaxonSDB**

Wie Sie sehen, wird hier gleichzeitig das Fenster für die Ein- und Ausgaben des debuggten Programms festgelegt.

2.5.3.7 Stackgröße

Wenn ein Programm aus MaxonPASCAL gestartet wird, braucht es natürlich auch einen Stack, und im Zweifelsfall nicht zu wenig davon. Die gewünschte Größe kann ebenfalls im „System"-Requester eingestellt werden.

2.5.3.8 Autosave

Wenn die ***„Autosave"***-Option vom MaxonPASCAL eingeschaltet ist, werden beim Start eines Programms aus der integrierten Umgebung automatisch alle Editor-Texte abgespeichert, sofern dies noch nicht geschehen ist - bestimmt eine sinnvolle Maßnahme. Dieser Autosave sollte nicht mit dem zeitabhängigen Autosave des Edward verwechselt werden.

Es gibt hier sogar gleich drei verschiedene Einstellungen:

- Aus:
  Der MaxonPASCAL-Autosave unterbleibt.
- Texte sichern:
  Die Texte im Editor werden nötigenfalls abgespeichert.
- Texte und Session:
  Zusätzlich werden die „SESSION"-Dateien von Edward und MPASCAL geschrieben. Dadurch haben Sie nach einem Absturz und erneutem Start von MPASCAL wieder alles so vor sich, wie es gewesen war.

2.5.3.9 Dateiauswahlbox

Der RCT-Filerequester hat seine Fans (mich eingeschlossen), aber Commodore sieht es gar nicht gern, wenn man gegen seine Richtlinien verstößt. Deshalb kann zwischen RCT- und ASL-Dateiauswahlbox frei gewählt werden.

2.5.4 Abspeichern der Konfigurationsdatei

2.5.4.1 Allgemeines

Wenn im aktuellen Verzeichnis keine Datei „mp.SESSION" gefunden wird, liest MaxonPASCAL die Einstellungen aus der Datei „S:mp.CONFIG" (siehe auch Abschnitt 1.2.3), und wenn es die auch nicht gibt, werden eben die internen Voreinstellungen verwendet. Mit dem Menüpunkt **„Optionen sichern"** werden alle aktuellen Einstellungen von MaxonPASCAL (ausgenommen natürlich der Editor-Zustand) in „S:mp.CONFIG" abgespeichert.

An dieser Stelle soll noch darauf eingegangen werden, wie diese Konfigurations- und Session-Dateien eigentlich aufgebaut sind. Zwar werden Sie darin kaum einmal etwas von Hand ändern, aber Sie müssen es wissen, wenn Sie über den ARexx-Befehl *„setopt"* MaxonPASCAL-Einstellungen ändern (siehe 2.6.3.6)

Die Konfigurationsdateien sind Texte mit einer beliebigen Folge von Optionen, die jeweils mit einem „-" beginnen.

2.5.4.2 Der Editor

Zunächst ist da ein Eintrag -X"Edward:Edward"

Wie Sie sich vielleicht denken können, entnimmt MaxonPASCAL dieser Angabe, woher der Editor Edward geladen werden soll. Falls Edward in Ihrem System nicht unter diesem Pfad und Namen zu finden sein sollte, können Sie diesen Eintrag ändern. Sie sollten aber nicht unbedingt versuchen, auf diese Weise einen anderen Editor als Edward einzubinden, denn nach derzeitigem Stand beherrscht kein fremder Editor das kompilzierte Message-Protokoll zwischen MaxonPASCAL und Edward. Alternative Editoren können ausschließlich über ARexx (Kapitel 2.6) integriert werden.

2.5.4.3 Fensterdaten

Am Anfang der Konfigurationsdatei finden Sie gewöhnlich eine ganze Reihe von Einträgen, die mit „-Xn" beginnen (wobei n eine Zahl zwischen 1 und 9 ist). Diese Daten legen Position und Größe der Fenster von MaxonPASCAL fest und sollten nie manuell verändert werden.

2.5.4.4 Switches und Schalter

Die Optionen „-X10" bis „-X18" repräsentieren diverse Schalterstellungen der integrierten MaxonPASCAL-Umgebung. Es folgen jeweils stets ein Komma und die Schalterstellung - meist „0" oder „1", manchmal aber auch höhere Nummern.

Und das bedeuten diese Einträge:

-X10: Exe-Datei automatisch erzeugen?
-X11: Objektdatei automatisch erzeugen?
-X13: Icons erzeugen?
-X14: Protokolldatei schreiben?
-X15: Autosave (0 = „Aus", 1 = „Texte sichern", 2 = „Texte + Session")
-X16: Größe des Laufzeitstacks
-X18: Dateiauswahlbox (0 = RCT, 1 = ASL)

2.5.4.5 Stringoptionen

Die Einträge „-X20" bis „-X23" geben den Inhalt von vier Stringgadgets wieder und erwarten folgerichtig als Argument eine „so" eingeschlossene Zeichenkette:

-X20: Name der Protokolldatei
-X21: Argumentstring bei Programmstart aus der Umgebung
-X22: Assembleraufruf
-X23: Debuggeraufruf

2.5.4.6 Compiler-Einstellungen

Die binären Compilerschalter werden wie folgt codiert:

Option	Aus	Ein
Subrange testen	**-gr**	**-g**
RIndexbereich	**-gi**	**-gI**
Unterbrechen	**-gb**	**-gB**
Stackgröße	**-gs**	**-gS**
Arithmet. Überlauf	**-gv**	**-gV**
Semikolon melden	**-ps**	**-pS**
Debugger-Dateien	**-b0**	**-b**
Make-Abhängigkeiten automatisch	**-mx**	**-mX**
Linker: Groß(klein egal	**-lc**	**-lC**

Der Compiler-Arbeistspeicher ist mit der Option „-w" einzustellen, also z. B. „-w200" für 200 KByte, und mit „-M" wird die Hauptdatei festgelegt. Mit den Optionen „-i" und „-u", jeweils gefolgt von einem Pfadnamen, werden die Pfade für Includes bzw. Units definiert. Die Pfade können, müssen aber nicht in Anführungszeichen eingeschlossen werden. Wollen Sie zugleich einen Kopierpfad angeben, so sind die beiden Pfade durch ein „|" voneinander zu trennen, z. B. „-i Pascal:include|ram:include". Last not least können Sie bei Bedarf auch mit der Option „-U", gefolgt von einem Unit-Namen, die Einstellung „Make-Units" vornehmen. Diese Option arbeitet allerdings inkrementell, d. h. jedes „-U" hängt einen weiteren Eintrag an, statt die bisherigen Einträge zu überschreiben. Eine Ausnahme bildet '-U "': Damit löscht man alle bisherigen Unitnamen. Letzteres wird zwar in den Konfigurationsdateien nicht benutzt, ist aber vor allem dann wichtig, wenn man über den ARexx-Port des Compilers Optionen verändern will (diese Anmerkung bezieht sich auf den setopt-Befehl aus Abschnitt 2.6.3.6 und braucht Sie im Moment noch nicht zu kümmern).

2.6 Die ARexx-Schnittstelle

2.6.1 Grundlagen

Die Programmiersprache Rexx ist in ihrer Amiga-Portierung „ARexx" zu beachtlicher Popularität gelangt, obwohl Rexx im Wirklichkeit doch eigentlich eher eine Script-Sprache und für „echte" Programme ziemlich unbrauchbar ist. Der Grund für diesen phänomenalen Erfolg ist (einmal abgesehen davon, daß ARexx ab Workbench 2.0 zum Lieferumfang jedes Amiga gehört), daß ARexx sich in hervorragender Weise dazu eignet, Programme „von außen" zu steuern und so zum einen komplexe Abläufe zu automatisieren und zum anderen wildfremde Programme zu einem System zu kombinieren.

Voraussetzung dafür ist, daß die Programme einen sogenannten ARexx-Port besitzen. Ein ARexx-Programm kann Anweisungen an eine solche Software-Schnittstelle schicken, und wenn das entsprechende Programm über einen geeigneten Befehlssatz verfügt, kann der Rexx-Programmierer Programmfunktionen mit einem ARexx-Script fernsteuern.

Der Editor „Edward" besitzt eine solche Schnittstelle, und da Edward Anweisungen an MPASCAL weitergeben kann, läßt sich die Compiler-Umgebung natürlich über diesen Umweg steuern.

Ein Beispiel: Das ARexx-Script

```
ADDRESS "EDWARD" 'SendAppComm "compile"'
```

veranlaßt MaxonPASCAL, den im Editor befindlichen Quelltext zu übersetzen. Voraussetzung dafür ist natürlich, daß MaxonPASCAL zuvor gestartet wurde. Mit der ARexx-Anweisung „*ADDRESS*" legt man übrigens fest, wie der Port heißt, mit dem das Script kommunizieren soll.

Es bleibt noch die Frage offen, was man macht, wenn man MaxonPASCAL ohne Edward benutzen und fernsteuern möchte - z. B. weil man einen anderen Editor bevorzugt (nahezu völlig unverständlich!) und diesen über ARexx in das MaxonPASCAL System integrieren möchte.

2.6.2 Der ARexx-Port

In Abschnitt 2.1 wurde bereits angedeutet, daß MPASCAL dafür vom CLI bzw. der Shell aus mit der Option „-r" zu starten ist, z. B. `MPASCAL -r`

In diesem Fall wird ein Port mit dem Namen „MAXONPASCAL" eingerichtet. Falls ein solcher schon existiert (kommt insbesondere vor, wenn MaxonPASCAL mehrfach gestartet wird), werden so lange „MAXONPASCAL1", „MAXONPASCAL2" usw. ausprobiert, bis ein noch nicht belegter Name gefunden wird. Alternativ können Sie auch hinter dem „-r" einen beliebigen Portnamen vorgeben, z. B.

```
MPASCAL -r "Bla Fasel Sülz!"
```

Generell werden solche Portnamen in Großbuchstaben umgewandelt. Auch hier werden systemweit eindeutige Namen sichergestellt und ggf. „BLA FASEL SÜLZ!1" usw. ausprobiert.

In jeden Fall startet nun MaxonPASCAL, meldet sich und gibt zur Kontrolle den Portnamen aus. Edward wird nicht geladen, MaxonPASCAL kommt im ARexx-Modus völlig ohne ihn aus.

Wenn Sie den Default-Namen verwenden, können Sie Ihr Script also locker-flockig mit ADDRESS „MAXONPASCAL" einleiten. Welche Anweisungen Ihnen dann zur Verfügung stehen, wird in den folgenden Abschnitten dargestellt. Zuvor sollten Sie aber noch erfahren, wie Sie das MaxonPASCAL wieder loswerden, ohne Ihren Rechner booten zu müssen. Zum einen gibt es dafür die Anweisung „*QUIT*". Das folgende kurze ARexx-Script entfernt also MPASCAL aus dem System:

```
Address "MAXONPASCAL" quit
```

An dieser Stelle sehen Sie auch, daß Groß-/Kleinschreibung von Befehlen in ARexx generell egal ist.

Andererseits reagiert MaxonPASCAL im ARexx-Modus auch auf die üblichen Abbruchsignale. Wenn Sie MPASCAL aus der Shell ohne „run" gestartet haben, genügt also die Tastenkombination <Ctrl>+<C>. Falls Sie „run" verwendet haben, können Sie mit der Shell-Anweiung „*status*" die Prozessnummer herausfinden und diesen Prozeß dann mit der „*break*"-Anweisung abmurxen.

2.6.3 Die ARexx-Befehle

2.6.3.1 Schließen, öffnen und Beenden

Wie bereits in 2.6.2 erläutert, kann MaxonPASCAL durch die Anweisung *quit* beendet werden. *shutdown* schlieflt alle Fenster, die MaxonPASCAL gerade geöffnet hat, z. B. weil das ganze System Iconifizert werden soll. Anschließend können mit *reopen* alle zuvor geöffneten Fenster erneut geöffnet werden.

2.6.3.2 Übersetzen, Starten und Ähnliches

Die ARexx-Befehle von MaxonPASCAL entsprechen im wesentlichen den Anweisungen, mit denen normalerweise Edward den Compiler steuert (also über *„SendAppComm"*) und die folglich auch in der „.DEF"-Datei verwendet werden. Allerdings haben einige Anweisungen hier Argumente, die zum Teil aber optional sind. So auch bei der wichtigsten Anweisung überhaupt: *„compile"*. *„compile hello.p"* veranlaßt MaxonPASCAL, das gewohnte Compiler-Fenster zu öffnen und die Quelldatei „hello.p" zu übersetzen. ARexx-Neulinge sollten beachten, daß die Anführungszeichen in diesem Fall notwendig sind, damit ARexx *„compile"* und „hello.p" als Teile einer zusammengehörenden Anweisung versteht und vollständig an MaxonPASCAL sendet.

„compile" ohne Dateinamen bewirkt die Compilation einer Datei namens „default.p" - falls vorhanden. Im Make-Modus (auch den gibt es natürlich im ARexx-Modus) werden die Dateinamen allerdings generell ignoriert und es wird stur nach der Make-Datei gearbeitet.

Die Anweisung *run* startet ein übersetztes Programm, falls ein solches vorliegt. Im Gegensatz zur „normalen" Entwicklungsumgebung wird dabei nicht automatisch compiliert, und es wird erst recht nicht geprüft, ob eine erneute Überstzung (z. B. wegen Veränderung von Quelltexten) nötig wäre.

Ein mit „run" gestartetes Programm kann mit der Anweisung *break* brutal abgebrochen werden. *debug* startet in gewohnter Weise den Debugger.

2.6.3.3 Requester und Informationen

Auch unter ARexx müssen Sie nicht darauf verzichten, die zahlreichen Optionen und Einstellmöglichkeiten des Systems über Dialogfenster zu wählen: *opsys* öffnet das System-Dialogfenster und *opcomp* öffnet das Dialogfenster für die Compiler-Optionen. Das Info-Fenster können Sie sich mit der Anweisung *info* ansehen, und nach einen Compilerdurchlauf läßt sich das bereits geschlossene Compiler-Fenster (in dem eventuell Fehlermeldungen stehen) mit der Anweisung *errors* erneut öffnen.

2.6.3.4 Make

Die Projektverwaltung steht auch im Rexx-Modus zur Verfügung. *„loadmk <Dateiname>"* lädt die angegebene Make-Datei. Wenn der Dateiname weggelassen wird, öffnet sich das Dateiauswahlfenster, und der Benutzer kann selbst eine Datei auswählen. Dagegen wird bei *newmk* stets eine vom Anwender zu benennende neue Projektdatei angelegt. Es ist nicht möglich, hier einen Dateinamen als Argument zu übergeben. *leavemk* vergißt die geladene Make-Datei und schaltet wieder in den Normal-Modus.

Zur Manipulation von Projekten stehen nur die beiden Funktionen zur Verfügung, die sich sinnvoll von einem Editor aus starten lassen: „*addtx <Quelltextname>*" fügt eine neue Quelltextdatei hinzu, und „*addtxobj <Quelltextname>*" trägt eine Quelltextdatei samt passender Objektdatei und geeigneter Compileranweiung in die Make-Datei ein. Dabei werden natürlich auch auf „.asm" endende Quelltexte sinnvoll behandelt.

Alle anderen Operationen kann der Anwender über das Make-Dialogfenster vornehmen. Es wird mit *openmake* geöffnet.

2.6.3.5 Schreiben von Dateien

Mit „*exefile <Dateiname>*" wird - falls ein übersetztes und gelinktes Programm vorliegt - eine ausführbare Programmdatei geschrieben, während „*objectfile <Dateiname>*" eine linkbare Objektdatei schreibt. Bei beiden Anweisungen ist der Dateiname optional. Wird er weggelassen, öffnet sich das Dateiauswahlfenster.

2.6.3.6 Optionen speichern und setzen

Die beiden Anweisungen „*saveconf <Dateiname>*" „*savesession <Dateiname>*" schreiben alle Einstellungen in die angegebene Datei. Der einzige Unterschied ist der Default-Name, den die Anweisungen verwenden, wenn kein Dateiname angegeben wird - dies sind nämlich „S:mp.config" und „mp.SESSION". Systemeinstellungen können aus einem ARexx-Script aber auch beliebig gesetzt werden. Dazu dient die Anweisung „*setopt <Optionen>*" Als Argument ist eine beliebig lange Folge von Einstellungen erlaubt. Diese sind in dem Format anzugeben, in dem sie in der Konfig- und in den Session-Dateien stehen.

Beispiel:

```
"setopt -X21,"hello World!" -pp -b -X10,1"
```

setzt den Argumentstring für Programmstarts auf „hello World!" und schaltet als Default in den Pascal-Modus, aktivert die Erzeugung von Debug-Dateien und läßt bei jeder Übersetzung automatisch eine Exe-Datei schreiben.

Alle Optionen, die die Entwicklungsumgebung betreffen, werden mit einer mit „-X" beginnenden Anweisung eingestellt und werden in Abschnitt 2.5.4 dieses Handbuchs erläutert.

2.6.3.7 Informationen über die Entwicklungsumgebung

Für elegante Lösungen sollten Scripts auch in der Lage sein, Informationen über den aktuellen Status von MaxonPASCAL zu erhalten. „*getinfo MAKE*" liefert den Namen der momentan geladenen Make-Datei bzw. einen Leerstring. Mit „*getinfo STATE*" erhalten Sie einen String „**NONE**", „**COMPILED**" oder „**LINKED**" und können so feststellen, ob ein übersetztes bzw. übersetztes und gelinktes Programm vorliegt. In beiden Fällen liegt er Ergebnisstring nachher in der Variablen „**RESULT**". Es ist auch zu beachten, daß vorher mit der ARexx-Anweisung *Options results* die Datenübergabe von MaxonPASCAL in das ARexx-Programm ermöglicht wurde.

2.6.3.8 Fehlermeldungen und Fehlerstellenansprung

Die Anweisung „*errorscript <dateiname>*“ legt ein ARexx-Script fest, das von MaxonPASCAL jedesmal aufgerufen wird, wenn der Compiler eine Fehlermeldung ausgibt. Diesem Script werden als Argumente Dateiname, Zeilen- und Spaltennummer sowie die eigentliche Fehlermeldung übergeben. So veranlaßt „*errorscript errors.rx*“ MaxonPASCAL bei jedem folgenden Compilerfehler zu einem Aufruf der Art:

```
rx errors.rx "hello.p" 8 1 "Syntax Error"
```

Ihr Script „errors.rx“ kann dann mit den übergebenen Daten beliebiges tun, z.B. sie in einer Fehlerdatei abspeichern.

Ein „errorscript“-Aufruf ohne Argument schaltet diesen Aufrufmechanismus wieder ab.

Da die Fehlerdaten schlicht an das Argument von „*errorscript*“ angehängt werden, können Sie trickreich auch weitere Argumente an das ARexx-Script übergeben:

```
"errorscript wagga.rx ERRORS arg1! bläk"
```

bewirkt Aufrufe wie

```
rx wagga.rx ERRORS arg1! bläk "hello.c" ...
```

Analog funktioniert die Anweisung „editscript <Scriptname>“, die dafür gedacht ist, auch im Rexx-Modus Fehlerstellenansprung zu realisieren. Das angegebene Script wird aufgerufen, wenn der Benutzer im Compiler-Fenster nach Ansprung einer Fehlerposition verlangt, und erhält als Argumente Dateinamen, Zeilen- und Spaltennummer, z. B.:

```
rx edit.rx "hello.p" 8 1
```

3. Der Spezialist für viel Tipparbeit

Durch die Beispiele im vorhergehenden Kapitel haben Sie sicher auch schon den neuen Editor von MaxonPASCAL kennengelernt. Einige Funktionen sind Ihnen bereits bekannt andere lassen sich unschwer erraten, wenn Sie mit der Maus in die Programm-Menüs blättern.

Im Kapitel 2.2 haben Sie bereits über den Preprozessor des Editors gelesen, der die sehr komfortable Konfiguration (Fehlermeldungen, Tastaturbelegungen, Menüs und die bedingten Menüs) des Editors aus einer Datei intepretiert. Die im folgenden beschriebenen Menüfunktionen, Menü-Tastaturkürzel und besondere Prüffunktionen sind Bestandteil der dem Paket beliegenden Konfigurationsdateien und können von Ihnen jederzeit auf spezielle Bedürfnisse angepaßt werden. Lesen Sie näheres zur grundlegenden Bedienung und zur individuellen Anpassung in den folgenden Kapiteln.

3.1 Installation von Edward auf der Festplatte

Edward wird beim Installieren des MaxonPASCAL-Compiler-Systems automatisch mitinstalliert.

3.2 Aufruf von Edward

Je nachdem, was Sie bevorzugen, läßt sich Edward sowohl aus dem CLI als auch von der Workbench aus starten. Für die fortgeschritteneren Anwendern ist hierbei vor allem interessant, welche Parameter schon beim Start an Edward übergeben werden können.

Hinweis

In Zusammenarbeit mit dem PASCAL-Compiler muß der Editor nicht extra gestartet werden. Dies übernimmt der Compiler für Sie. Lesen Sie die beiden folgenden Abschnitte, wenn Sie EDWARD auch unabhängig von MaxonPASCAL anwenden möchten.

3.2.1 Start von der Workbench

Am einfachsten starten Sie Edward durch Doppelklick auf sein Icon. Dieses befindet sich nach einer normalen Installation im Verzeichnis „Edward:" auf der von Ihnen gewählten Partition Ihrer Festplatte oder auf der Diskette.

Sie können Edward ebenfalls starten, wenn Sie auf das Icon einer mit Edward bearbeiteten Datei geklickt haben.

Hinweis

Wie Sie Icons zu einer Datei erzeugen lassen können und was es dabei zu beachten gilt, lesen Sie im Kapitel 3.15 „Globale Einstellungen", insbesondere sollten Sie hier die Bemerkungen zum „Icon Tool" beachten.

Wenn Sie mit gedrückter Shift-Taste die Icons verschiedener Dateien anwählen und daraufhin Edward durch Doppelklick starten (Shift-Taste gedrückt lassen), so lädt Edward automatisch die von Ihnen angewählten Dateien ein. Angezeigt wird Ihnen allerdings nur die letzte geladene Datei.

☞ Zum gleichzeitigen Bearbeiten verschiedener Dateien lesen Sie mehr im Kapitel 3.3 „Text".

3.1.2 Start aus dem CLI

Normalerweise starten Sie Edward von CLI aus durch einfache Eingabe von „Edward". Falls Ihnen das zu lang ist, so benennen Sie Edward einfach um oder Sie benutzen den „Alias"-Befehl Ihrer Shell (Mehr dazu lesen Sie in der von Commodore ausgelieferten Dokumentation oder in den entsprechenden HotHelp-Projekten). Sie können selbstverständlich Namen von Dateien angeben, die nach dem Start von Edward automatisch geladen werden sollen.

Nach dem Start von Edward aus dem CLI erscheint die Meldung „Edward Process successfully detached". Das bedeutet, daß Edward von nun an unabhängig vom CLI als eigener Prozeß läuft. Sie können weiter mit dem CLI arbeiten, ohne daß dieser blockiert ist und können den CLI auch jederzeit beenden.

3.2 Eingabe und Bearbeitung von Texten

Zu Beginn erläutere ich kurz die Bedeutung der Status-Anzeige, danach werde ich Ihnen die voreingestellte Tastaturbelegung erklären, so daß Sie alle üblichen Operationen bei der Eingabe eines Textes vornehmen können. Desweiteren werde ich noch zusätzliche Möglichkeiten vorstellen, wie Sie die Cursor-Position im Text verändern können.

3.2.1 Die Status-Anzeige

Wichtige Informationen über den gerade bearbeiteten Text erhalten Sie auf einen Blick, wenn Sie die Titelzeile eines momentan aktiven Text-Fensters von Edward betrachten.

Ganz links erscheint dort immer der Name des gerade bearbeiteten Textes. Anschließend folgen zwei Zahlenangaben, die erste gibt die aktuelle Zeile an, die zweite die Spaltenposition innerhalb dieser Zeile.

Anschließend folgen fünf durch Komma getrennte Buchstaben. Diese geben an, ob bestimmte Optionen für den aktuellen Text aktiv sind, dann ist der entsprechende Buchstabe groß, oder nicht (der Buchstabe ist dann logischerweise klein).

Das **„c"** ganz links gibt an, ob der Text seit dem Einladen oder nach dem letzten Abspeichern verändert wurde (großes **„C"** falls dies der Fall ist).

Das **„f"** gibt an, daß Falten geändert wurden.

Ein großes **„W"** bedeutet, daß der Text im „Word-Wrap"-Modus bearbeitet wird.

Ein **„O"** gibt an, daß nun der „OverWrite"-Modus gilt und ein großes **„L"** signalisiert, daß der Text momentan ge-"locked" ist. Ge-"locked" bedeutet, der Text kann momentan nicht verändert werden. Dies ist zum Beispiel der Fall, wenn Sie den entsprechenden Text drucken lassen.

☞ Mehr zu den oben erwähnten Modi lesen Sie in den Kapiteln 3.8 „Das Edit-Prefs Formular" und 3.9 „Word-Wrap".

3.2.2 BACKSPACE und DELETE

Die Tasten <BACKSPACE> und <DELETE> sind wie üblich belegt. Mit <CTRL>+<DELETE> können Sie die komplette aktuelle Zeile löschen.

3.2.3 UndoLine-Funktion

Edward bietet die Möglichkeit das Löschen einer Zeile oder Änderungen einer Zeile rückgängig zu machen, falls Sie nicht schon eine weitere Zeile verändert haben.

- Geben Sie zum Test folgendes ein : „Der dumme Detlev demoliert die Damentoilette". Fahren Sie nun mit dem Cursor auf „Detlev" und ändern Sie den Namen in „Dödel". Drücken Sie nun <CTRL>+<U>. Daraufhin erscheint statt „Dödel" wieder „Detlev".
- Löschen Sie nun die komplette Zeile mit <CTRL>+<DELETE>. Mit <CTRL>+<U> erhalten Sie die komplett gelöschte Zeile wieder zurück.

3.2.4. RETURN, ENTER und TAB

Falls Sie <RETURN> drücken und der Cursor befindet sich innerhalb einer Zeile, so wird die Zeile an dieser Position aufgespalten. Ähnliches gilt für <ENTER>. Drücken Sie <RETURN> oder <ENTER> am Anfang einer Zeile, so wird vor dieser eine leere Zeile eingefügt. Drücken Sie <RETURN> oder <ENTER> am Ende einer Zeile, so wird nach dieser Zeile eine leere Zeile eingefügt. Der Unterschied zwischen <RETURN> und <ENTER> ist, daß <ENTER> die neu erzeugte Zeile entsprechend der vorigen Zeile einrückt. Dies ist ein besonders für Programmierer nützliches Verhalten, das das strukturierte Programmieren unterstützt.

Mit der <TAB>-Taste fügen Sie ein Zeichen in den Text ein, daß den Platz von 3 Leerzeichen einnimmt. Dies dient ebenfalls der besseren Strukturierung von Texten und Programmen.

☞ Die Veränderung der TAB-Weite wird im Kapitel 3.8 über das Edit-Prefs-Formular erklärt.

Um die oben erklärten Funktionen von <TAB> und <ENTER> zu testen, können Sie folgendes ausprobieren.

- Drücken Sie dreimal die <TAB>-Taste. Wie Sie an der rechten Zahl in der Titel-Leiste des Fensters sehen, befindet sich der Cursor nun an der Position 10 im Text. Geben Sie nun `FOR v := e1 TO e2 DO s;` oder was Ihnen beliebt ein.
- Drücken Sie nun die <ENTER>-Taste. Sie befinden sich nun in einer neuen Zeile und der Cursor befindet sich wieder auf Position 10, also auf derselben Einrückungsstufe wie die vorige Zeile.

3.2.5 Die Cursor-Tasten und der Ziffernblock

Wenn Sie die CURSOR-Tasten ganz normal drücken, also nicht in Kombination mit <ALT>, <CTRL> oder <SHIFT>, so bewegen Sie den Cursor wie gewohnt. Dies ist ebenfalls mit den Tasten des Ziffernblock möglich. Diese müssen Sie allerdings in Kombination mit der <ALT>-Taste drücken (ALT- 8, 4, 6, 2).

In Verbindung mit <ALT> bewegen Sie sich mit <CURSOR-Left> und <CURSOR-Right> um ein Wort vorwärts oder rückwärts. Mit <CURSOR-Up> und <CURSOR-Down> geht es jeweils vier Zeilen nach oben bzw. unten.

Mit <SHIFT> bewegen Sie den Cursor mit <CURSOR-Left> und <CURSOR-Right> an den Anfang bzw. das Ende der Zeile. In Verbindung mit <CURSOR-Up> und <CURSOR-Down> bewegen Sie sich jeweils um fast eine Bildschirmseite nach oben bzw. unten.

Mit <ALT-PgUp> (ALT-9) und <ALT-PgDn> (ALT-3) des Ziffernblocks bewegen Sie den Cursor auf dieselbe Weise.

Mit <ALT-Home> (ALT-7) und <ALT-End> (ALT-1) des Ziffernblock springen Sie an den Anfang bzw. das Ende des Textes.

Benutzen Sie die Tasten <8> und <2> des Ziffernblocks in Kombination mit der linken <AMIGA-Taste>, so können Sie den Text um jeweils eine Zeile nach oben oder unten scrollen. Dabei bleibt der Cursor an seiner Position, und der Text verschiebt sich unter ihm in beide Richtungen.

3.2.6 Der Schieberegler

Sie können die Position des Cursors im Text ebenfalls mit dem Slider an der rechten Seite des Textfenster bestimmen. Wenn Sie vor oder hinter den Slider klicken, bewegen Sie den Cursor immer um eine Seite vor oder zurück.

3.2.7 Die Maus

Es ist zusätzlich möglich, den Cursor mit der Maus an eine bestimmte Position zu bewegen. Dazu bewegen Sie einfach den Mauspfeil auf die gewünschte Position und drücken die linke Maustaste.

Solange Sie die linke Maustaste gedrückt halten, folgt der Cursor dem Mauspfeil. Bewegen Sie sich an den oberen oder unteren Rand des Fensters, so scrollt der Text in die entsprechende Richtung. Die Geschwindigkeit, mit welcher dieser Vorgang abläuft, hängt davon ab, ob sich der Mauspfeil ganz rechts - dies entspricht einer sehr langsamen Bewegung - oder ganz links - dies entspricht einer sehr schnellen Bewegung, - befindet.

3.3 Text

Ein Text besteht einfach aus einer oder mehreren Zeilen, die Sie eingeben oder als Datei von Diskette oder Festplatte geladen haben.

☞ Das Laden von Dateien (=Texten) finden Sie im Kapitel 3.4 „Dateioperationen."
Mit Edward können Sie fast beliebig viele Texte im Speicher halten, zwischen diesen umschalten und (natürlich :-)) die Texte bearbeiten.

3.3.1 Erzeugen von Texten

- Starten Sie bitte den Pascal-Compiler oder einfach nur den Edward.

Edward hat automatisch alles Nötige unternommen, damit Sie sofort einen Text oder ein Programm eingeben können.

- Tippen Sie nun – *Mein 1. Text mit Edward* – ein.
- Wir werden jetzt Edward mitteilen, daß wir einen weiteren Text bearbeiten wollen. Dazu wählen Sie aus dem „Text"-Menü den Eintrag **„Neu"**. Das Fenster von Edward ist daraufhin wieder leer. Das ist richtig, denn Sie befinden sich jetzt im zweiten Text, für den Sie eben Speicher reserviert haben.
- Tippen Sie nun – *Mein 2. Text mit Edward* – ein.

Mit dem oben beschriebenen Menüpunkt können Sie jederzeit Edward mitteilen, daß Sie einen weiteren Text bearbeiten wollen. Verfahren Sie wie oben beschrieben und erzeugen Sie einen dritten Text und geben daraufhin – *Mein 3. Text mit Edward* – ein.

3.3.2 Wahl eines bestimmten Textes

Um wieder zurück zum ersten Text zu gelangen, gibt es verschiedene Möglichkeiten.

- Wählen Sie den Eintrag **„Vorheriger"** aus dem Text-Menü. Edward zeigt Ihnen nun die Zeile „Mein 2. Text mit Edward" an. Mit **„Vorheriger"** wählen Sie immer den vorigen Text aus.
- Wählen Sie erneut den Eintrag **„Vorheriger"** aus dem Text-Menü. Edward zeigt Ihnen nun die Zeile „Mein 1. Text mit Edward" an. Wir befinden uns also jetzt, wie gewünscht, wieder im ersten Text.

- Wählen Sie noch einmal den Eintrag **„Vorheriger"** aus dem Text-Menü. Edward zeigt Ihnen die Zeile „Mein 3. Text mit Edward" an. Vor dem ersten Text kann natürlich kein weiterer liegen. Darum wählt Edward in diesem Fall den letzten Text. Sie können also mit **„Vorheriger"** die Texte der Reihe nach anwählen.

Genauso können Sie durch Wählen von **„Nächster"** im „Text"-Menü den folgenden Text anwählen. Haben Sie gerade den letzten Text bearbeitet, so wählt Edward in diesem Fall den ersten Text.

- Eine direkte Wahl des Textes ist ebenfalls möglich. Möchten Sie den zweiten Text bearbeiten, dann wählen Sie einfach **„Text 2"** aus dem „Text"-Menü.

3.3.3 Freigeben eines Textes

Nach der Bearbeitung eines Textes (und dem Abspeichern desselben) können Sie Edward mitteilen, daß Sie diesen Text freigeben möchten.

- Wählen Sie **„Löschen"** aus dem „Text"-Menü. Edward zeigt Ihnen daraufhin die Zeile „Mein 3. Text mit Edward" an. Der zweite Text wurde von Edward freigegeben. Nach der Freigabe eines Textes versucht Edward, den folgenden zu wählen. Haben Sie den letzten Text freigegeben, dann wählt Edward den vorigen Text für die weitere Bearbeitung aus.

Hinweis

Sollten Sie aus Versehen die Freigabe eines Textes eingeleitet haben, der noch nicht abgespeichert wurde, so erscheint ein Dialogfenster mit der Frage, ob Sie diesen Text „vergessen" wollen. Geben Sie „Nein" an, so wird die Operation abgebrochen, wählen Sie „Ja", dann gehen alle Änderungen, die an diesem Text vorgenommen wurden, verloren!

3.3.4 Löschen des aktuellen Textes

Mit dem Eintrag **„Entfernen"** aus dem „Text"-Menü können Sie alle Zeilen des aktuellen Textes löschen und danach einen neuen Text eingeben. Der Text wird dadurch aber nicht freigegeben.

3.3.5 Drucken eines Textes

Mit dem Eintrag **„Drucken"** aus dem „Projekt"-Menü teilen Sie Edward mit, daß er den Text, den Sie momentan bearbeiten, ausdrucken soll. Während Edward diesen Text druckt, können Sie weiterhin andere Texte bearbeiten und desweiteren alle Funktionen von Edward benutzen, die den aktuellen Text nicht verändern. Sie können also, während Edward Ihren Text druckt, weiterhin in diesem den Cursor positionieren oder z.B. Blöcke markieren und kopieren. Das Einfügen oder Ausschneiden von Blöcken ist dagegen nicht möglich.

Während des Druckvorgangs erscheint ein kleines Fenster. Wenn Sie den Druckvorgang unterbrechen wollen, so schließen Sie einfach dieses Fenster.

☞ Das Format, in welchem eine Textseite gedruckt werden soll, können Sie auf vielfältige Weise variieren. Lesen Sie dazu mehr im Kapitel 3.15 „Globale Einstellungen".

Hinweis

Edward gibt den Text an den Drucker aus, den Sie mit den Preferences der Workbench gewählt haben. Sollten Sie hiermit nicht klarkommen, so schauen Sie im Handbuch zur Workbench nach, das zu jedem Amiga mitgeliefert wird.

3.4 Dateioperationen

Hinter dem hochtrabenden Titel **„Dateioperationen"** verbergen sich Funktionen zum Abspeichern und Laden von Texten.

3.4.1 Speichern von Texten

Nachdem Sie im ersten Kapitel gelernt haben, wie man einen Text eingibt und bearbeitet, zeige ich Ihnen nun, wie man einen solchen Text auf Diskette oder Festplatte abspeichert und wie man den Text von dort zur Bearbeitung wieder in den Editor bekommt.

- Geben Sie nun einen beliebigen Text ein, z.B.: *„Alles Seiende hat sein Sein im Nicht-Sein seines Gegensatzes."*
- Wählen Sie aus dem „Projekt"-Menü den Eintrag **„Speichern als..."**. Es erscheint daraufhin ein Dateiauswahlfenster, in dessen Titelzeile „Text speichern" steht. Speichern Sie den Text unter dem Namen **„Test.TXT"** ab.

Normalerweise wird beim Abspeichern eines Textes für diesen ein Pictogramm erzeugt, so daß er auch auf der Workbench sichtbar wird. Desweiteren werden einige Zusatzinformationen, die den Text betreffen mitabgespeichert (unter anderem die Tabulatorweite, Textfalten, etc.). Besonders interessant ist hierbei, daß Edward auch die aktuelle Cursorposition speichert. Das bedeutet, wenn Sie einen Text erneut einladen, dann befindet sich der Cursor an derselben Stelle, an der Sie zuletzt gearbeitet haben.

☞ Das Erzeugen des Pictogrammes und das Speichern der Zusatzinformationen können Sie natürlich auch abstellen. Lesen Sie hierzu das Kapitel 3.15 „Globale Einstellungen".

Hinweis

Unter Umständen erscheint ein „System Request" mit dem Hinweis, das Ihre Diskette schreibgeschützt ist. Sorgen Sie dann dafür, das der Schreibschutz entfernt wird oder nehmen Sie eine andere Diskette.

Wichtig

Die Bedeutung des Abspeicherns kann gar nicht oft genug betont werden. Alle Veränderungen, die Sie seit dem letzten Abspeichern vorgenommen haben, sind verloren, wenn ein System-Absturz auftritt, der Strom ausfällt oder sonstige unvorhersehbare Ereignisse eintreten. Sie können durch eine spezielle Einstellung dafür sorgen, daß Edward in bestimmten Zeitabschnitten Ihre Texte automatisch abspeichert.

☞ Prefs/Autosave

3.4.2 Laden von Texten

Nachdem Sie nun einen Text abgespeichert haben, zeige ich Ihnen im Folgenden wie Sie diesen Text wieder einladen.

- Wählen Sie aus dem „Text"-Menü den Eintrag **„Entfernen"**. Das Textfenster ist nun leer. Alle Textzeilen wurden freigegeben. Dies ist nur für unser Beispiel notwendig.
- Wählen Sie nun den Eintrag **„Öffnen"** aus dem „Projekt"-Menü. Es erscheint wie vorher beim Abspeichern der Dateiauswahlfenster. In diesem Fall steht in seiner Titelzeile aber „Text laden". Laden Sie den Text **„Test.TXT"** wieder ein.

Hinweis

Wenn Sie einen Text einladen, wird dieser über den Inhalt Ihres aktuellen Textes geladen. Wenn Sie dies vermeiden wollen, so sollten Sie zuerst einen neuen Text erzeugen, bevor Sie **„Öffnen"** wählen. Sollte der aktuelle von Ihnen bearbeitete Text noch nicht gespeichert worden sein, so erscheint ein Dialog-Fenster, das Sie auf diesen Umstand aufmerksam macht und mit dem Sie das Überladen dieses Textes verhindern können.

3.5. Blockoperationen

Größere Teile eines Textes umgruppieren oder vervielfältigen zu können, ist eine der wichtigsten Fähigkeiten eines Editors. Edward bietet Ihnen auch auf diesem Gebiet alles Nötige und einiges darüber hinaus.

Ein besonders markierter Bereich eines Textes wird im allgemeinen als „Block" bezeichnet. Daraus leitet sich auch der Name dieses Kapitels ab.

3.5.1 Cut, Copy, Paste

Hinter den Begriffen **„Ausschneiden"**, **„Kopieren"** und **„Einfügen"** verbergen sich die elementaren Blockoperationen.

Mit **„Ausschneiden"** schneiden Sie einen markierten Block aus Ihrem Text aus und speichern ihn in einem Zwischenspeicher ab. **„Kopieren"** kopiert den Block nur in den Zwischenspeicher, ohne ihn auszuschneiden und **„Einfügen"** fügt den Block ab der aktuellen Cursor-Position in den Text ein.

Um eine Blockoperation ausführen zu können, müssen Sie zuerst einmal einen Block markieren. Dies geschieht indem Sie **„Markieren"** aus dem „Block"-Menü wählen. Wenn Sie nun den Cursor bewegen, so werden Sie feststellen, daß der gesamte Textbereich zwischen der Position an der sich der Cursor momentan befindet und dem Ort, an dem sich der Cursor befand, bevor Sie **„Markieren"** gewählt haben, invertiert dargestellt wird. Dieser invertiert dargestellte Bereich ist der sogenannte markierte „Block". Alle Blockoperationen wirken sich auf diesen Bereich aus.

Wenn Sie erneut **„Markieren"** wählen, wird der gesamte Text wieder normal dargestellt. Der markierte Block wurde damit vergessen.

- Für die folgenden Schritte sollten Sie nun einen etwas längeren Text einladen.
- Positionieren Sie nun den Cursor in die Mitte irgendeiner Zeile am Anfang des Textes und wählen Sie danach **„Markieren"** aus dem „Block"-Menü.
- Fahren Sie nun mit dem Cursor in irgendeine Zeile, die weiter unten liegt. Sie sehen nun einen invertiert dargestellen Bereich.
- Wählen Sie **„Ausschneident"** aus dem „Block"-Menü. Der markierte Bereich verschwindet daraufhin. Der Cursor befindet sich nun auf derselben Position, an der er sich befand, bevor Sie **„Markieren"** gewählt hatten. Rechts vom Cursor befindet sich der Rest der Zeile, auf der sich der Cursor befand, bevor Sie **„Ausschneiden"** gewählt hatten.

Um nun die ursprüngliche Gestalt des Textes wiederherzustellen, können Sie den ausgeschnittenen Bereich wieder in den Text einfügen. Dazu dürfen Sie den Cursor nicht bewegen.

- Wählen Sie **„Einfügen"** aus dem „Block"-Menü. Der Cursor befindet sich nun am Ende des eingefügten Bereiches. Ihr Text hat wieder seine ursprüngliche Gestalt.

Sie können jederzeit an anderen Positionen im Text eine Kopie des ausgeschnittenen Bereiches einfügen, indem Sie einfach **„Einfügen"** anwählen.

Wenn Sie einen bestimmten Textbereich nur kopieren wollen, so markieren Sie den Bereich, wählen **„Kopieren"**, positionieren den Cursor an die gewünschte Stelle und wählen danach **„Einfügen"**.

☞ Zum Zwischenspeichern von Textausschnitten bedient sich Edward des Zwischenspeichers. Dies ist eine genormte Ablage, der sich alle Programme, die für den Amiga entwickelt werden, bedienen sollten. Falls Sie zum Beispiel ein Desktop-Publishing Programm besitzen, aber nicht auf die komfortablen Editier-Möglichkeiten Edwards verzichten wollen, so können Sie sich des Zwischenspeichers bedienen, um Texte zwischen Edward und dem DTP-Programm auszutauschen. Dabei ist natürlich Voraussetzung, daß das DTP-Programm ebenfalls den Zwischenspeicher unterstützt.

3.5.2 Spaltenblöcke

Eine Besonderheit im Umgang mit Blöcken bietet Ihnen Edward mit den Spaltenblöcken. Haben Sie zum Beispiel eine Tabelle mit mehreren Spalten eingegeben und Sie wollen nun eine der Spalten kopieren oder vertauschen, dann bewerkstelligen Sie dies unter Zuhilfenahme von Edwards Spaltenblöcken.

- Geben Sie folgende Tabelle ein

Film	Bewertung	Hauptdarsteller	!Test
Alien I	1-	Sigourney Weaver	!
Alien II	1	Sigourney Weaver	!
Terminator II	1++	Arnold Schwarzenegger	!
Homo Faber	1	Sam Shephard	!
Gefährliche Liebschaften	1+	John Malkovic,	!
		Glenn Close,	!
		Michelle Pfeiffer	!
The Fabulous Baker Boys	1	Michelle Pfeiffer,	!
		Jeff & Beau Bridges	!
Das Russlandhaus	2	Sean Connery,	!
		Michelle Pfeiffer	!
Harry & Sally	1	Meg Ryan	!
Night on Earth	1	Wynona Ryder	!
Fessle mich!	1	-	!
Das Wunderkind Tate	2+	Jodie Foster	!
König der Fischer	2+	Robin Williams	!
Hook	6	Dustin Hoffmann,	!
		Robin William,	!
		Julia Roberts	!

Sie wollen nun die Spalte „Bewertung" mit der Spalte „Hauptdarsteller" vertauschen.

- Fahren Sie mit dem Cursor auf das B von „Bewertung". Wählen Sie dann **„vertikal markieren"** aus dem „Block"-Menü.
- Bewegen Sie nun den Cursor vor das R von „Robin Williams" in der letzten Zeile, Spalte „Hauptdarsteller". Wählen Sie anschließend **„Ausschneiden"** aus dem „Block"-Menü. Die Spalte „Bewertung" ist nun verschwunden, „Hauptdarsteller" befindet sich an deren Stelle.
- Positionieren Sie den Cursor auf das Ausrufungszeichen in der ersten Zeile der Spalte „Test". Wählen Sie nun **„vertikal einfügen"** aus dem „Block"-Menü.

Daraufhin erscheint die Spalte „Bewertung" wieder. Die Vertauschung hat funktioniert.

3.5.3 Laden, Speichern und Drucken

Sie können einen mit **„Ausschneiden"** oder **„Kopieren"** in die Zwischenablage kopierten Block auf einfache Weise als Datei abspeichern. Dazu wählen Sie einfach „Block speichern" aus dem

„Block"-Menü. Es erscheint der Dateiauswahlfenster und Sie können einen Namen für den Block angeben.

Wollen Sie einen so abgespeicherten Block wieder in Ihren Text einfügen, so wählen Sie **„Block einlesen"** aus dem „Block"-Menü. Mit dem Dateiauswahlfenster geben Sie eine Datei an. Daraufhin wird die angebene Datei in den Zwischenspeicher geladen. Sie können sie dann jederzeit mit **„Einfügen"** an einer beliebigen Stelle im Text einfügen.

Mit dem Menüpunkt **„Block drucken"** aus dem „Block"-Menü können Sie einen in die Zwischenablage gespeicherten Block ausdrucken, allerdings ohne die Ausgabe besonders zu formatieren, wie es beim Ausdruck eines ganzen Textes geschieht.

3.5.4 Einrücken

Zur Strukturierung von Texten, insbesondere von Programmen, dient das Einrücken von Textabschnitten, die damit als zusammengehöriger Abschnitt hervorgehoben werden sollen. Zur Unterstützung dieser Maßnahmen bietet Ihnen Edward die Möglichkeit, einen markierten Bereich von Zeilen einzurücken.

- Geben Sie folgendes Programmfragment ein:

```
WHILE count >= 0 DO
BEGIN
s;
v:=succ(v);
count:=count -1;
END;
```

Um den Bereich zwischen BEGIN und END besser hervorzuheben, können Sie ihn einrücken.

- Bewegen Sie dazu den Cursor auf die Zeile nach BEGIN und wählen Sie **„Markieren"** aus dem „Edit"-Menü.
- Fahren Sie nun mit dem Cursor an das Ende der Zeile vor END.

Mit <CTRL>+<CursorLeft> und <CTRL>+<CursorRight> können Sie nun den markierten Bereich um jeweils einen TAB-Schritt nach rechts oder links verschieben.

- Drücken Sie <CTRL>+<CursorRight>. Damit haben Sie den Block um eine TAB-Position nach rechts verschoben. Probieren Sie ruhig ein bißchen die Möglichkeiten aus, die sich mit dem oben beschriebenen Verfahren ergeben.

3.5.5 Umwandeln in Groß- bzw. Kleinbuchstaben

Mit zwei weiteren Funktionen können Sie alle Buchstaben eines markierten Blocks in Groß- bzw. Kleinbuchstaben wandeln lassen.

- Geben Sie „Alles Seiende entsteht ohne Grund, setzt sich aus Schwäche fort und stirbt durch Zufall." oder etwas was Ihnen mehr zusagt, ein.
- Markieren Sie einen Block, der vom ersten Buchstaben von „Seiende" bis zum letzten Buchstaben von „aus" reicht.
- Wählen Sie nun **„Großbuchstaben"** aus dem „Extras"-Menü. Augenblicklich werden alle markierten Buchstaben in Großbuchstaben umgewandelt.

Wenn Sie **„Kleinbuchstaben"** aus dem „Extras"-Menü wählen, so werden entsprechend alle Buchstaben des markierten Blocks in Kleinbuchstaben gewandelt.

3.6 Suchen und Ersetzen

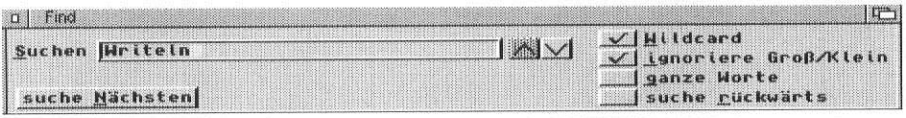
Suchen und Ersetzen von Zeichenfolgen in einem Text gehören zum Standardrepertoire eines Editors. Hier läßt Sie Edward natürlich nicht im Stich.

Um Ihnen das Arbeiten beim Suchen und Ersetzen zu erleichtern, bietet Edward zwei Extra-Fenster mit diversen Einstellmöglichkeiten an, über welche Sie die Arbeiten beim Suchen und Ersetzen steuern. Da Sie in diesen Fenstern gewisse Eintragungen machen können, sprechen wir im folgenden von sogenannten „Formularen".

3.6.1 Das „Find"-Formular

Zum Suchen eines bestimmten Begriffes bedienen Sie sich des „Find"-Formulars.

Dieses erscheint, wenn Sie im Menü „Suchen" den Menüpunkt **„Suchen ..."** wählen.



Wir werden nun gemeinsam die Funktionen des „Find"-Formulars durchsprechen.

Suchen

In diesem Bereich geben Sie die zu suchende Zeichenfolge ein. Nach dem Öffnen eines Find-Formulars ist das Eingabefeld automatisch aktiv und zur Eingabe bereit. Wenn Sie die Eingabe mit RETURN abschliessen, so verschwindet das „Find"-Formular gleich darauf und der Cursor wird auf die erste Stelle im Text positioniert, an der diese Zeichenfolge gefunden werden konnte.

Für die Suche einer Zeichenfolge gilt es dabei Folgendes zu beachten:

Gesucht wird immer von der aktuellen Cursorposition aus. Normalerweise wird von dieser ausgehend dann in Richtung auf das Ende des Textes hin gesucht. Konnte dabei keine Zeichenfolge im Text gefunden werden, die der gewünschten entspricht, so wird die Suchrichtung umgekehrt. Jeder weitere Suchvorgang geht dann von der Cursorposition aus

zum Beginn des Textes hin. Wird dort nichts mehr gefunden, so kehrt sich die Richtung erneut um.

mehrere Suchbegriffe

Eine Besonderheit bietet Edward Ihnen mit den beiden Knöpfen rechts neben dem Texteingabefeld. Sie können insgesamt 10 verschiedene Suchmuster eingeben. Mit den Pfeilen können Sie zwischen Ihnen umschalten.

Wildcard

Hiermit können Sie beim Suchen sogenannte WildCards eingesetzen. Als sog. Jokerzeichen dient dabei der Stern „*". An jeder Stelle an der sich in der zu suchenden Zeichenfolge ein Stern befindet darf beim Vergleich mit einer anderen Zeichenfolge eine beliebige Teilfolge stehen.
Geben sie zum Beispiel „Co*m*e" als zu suchende Zeichenfolge ein, so wird beim Vergleich unter anderem „Commodore" oder „Commode" als gefunden erkannt.

ignoriere Groß/Klein

Wenn Sie „ignoriere Groß/Klein" anwählen, so wird beim Suchen nicht zwischen Groß- und Kleinbuchstaben unterschieden.

ganze Worte

Ist dieses Feld angewählt, so wird nur nach vollständigen Wörtern gesucht. Wenn Sie also die Zeichenfolge „Kohl" suchen, so gilt „Kohlsuppe" oder „Kohldampf" als nicht gefunden. Nur „Kohl" wird gefunden und der Cursor entsprechend positioniert.

suche rückwärts

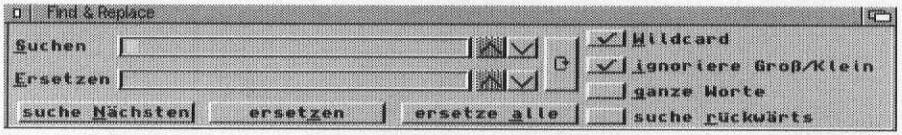
Hiermit können Sie die Suchrichtung jederzeit umkehren.

Suche nächsten

Mit Klick auf dieses Feld positionieren Sie den Cursor automatisch auf das nächste Vorkommen der angebenen Zeichenkette.

3.6.2. Das „Find & Replace"-Formular

Mit dem „Find & Replace"-Formular können Sie auf komfortable Weise Zeichenfolgen im Text suchen und durch andere Zeichenfolgen ersetzen lassen.



Die Bedienung des „Find & Replace"-Formular ist mit der des „Find"-Formular weitgehend identisch.

Suchen

Geben Sie in dieses Textfeld die zu suchende Zeichenfolge an.

Ersetzen

und in „Ersetzen" die Zeichenfolge durch welche ersetzt werden soll. Durch das Öffnen des „Find & Replace"-Formulars wird automatisch das Eingabefeld „Suchen" aktiviert. Nach der Eingabe in „Suchen" wird automatisch „Ersetzen" aktiviert.

mehrere Such-/Ersetzbegriffe

Mit den Knöpfen rechts neben „Suchen" und „Ersetzen" können Sie zehn verschieden Suchmuster anwählen, mit zehn verschieden Zeichenfolgen durch die diese ersetzt werden sollen.

Vertauschen der Such-/Ersetzbegriffe

Als weitere Komfortsteigerung können Sie mit Klick auf den Knopf rechts daneben die Such- und Ersetzungszeichenfolgen miteinander vertauschen. Falls Sie also aus Versehen zum Beispiel „Hund" suchen und durch „Katze" ersetzen ließen so können sie mit Hilfe dieser Funktion das Ganze rückgängig machen.

suche nächsten

Mit Klick auf dieses Feld positionieren Sie den Cursor automatisch auf das nächste Vorkommen der in „Suchen" angebenen Zeichenkette.

ersetzen

Durch Klicken auf dieses Feld wird von der aktuellen Cursor-Position aus die in „Suchen" angebebene Zeichenfolge gesucht und, falls gefunden, durch die in „Ersetzen" angebenene Zeichenfolge ersetzt. Danach wird automatisch das nächste Vorkommen des Begriffes in „Suchen" gesucht und der Cursor darauf positioniert.

ersetze alle

Hiermit werden im gesamten Text automatisch alle Vorkommen des in „Suchen" angegebenen Textes durch denin „Ersetzen" angegebenen Textes ersetzt. Anschließend wird der Cursor auf den Beginn des Textes positioniert.

3.6.3 Struktur-Check (Klammer-Check)

Für Programmierer (insbesondere diejenigen, die die Sprache „C" benutzen) kann der Strukur-Check sehr hilfreich sein. Mit diesem läßt sich auf einfache Weise prüfen, ob für jede öffnende Klammer in einem Ausdruck auch eine schließende Klammer vorhanden ist. Dabei werden nicht nur die gewohnten runden Klammern „(" und „)" überprüft, sondern auf Wunsch auch „[" und „]" sowie „{" und „}".

- Geben Sie folgende Zeilen ein :

```
void main( void )
{
   int i;
   for( i=100; i>0; i--)
   { f( i & 1 )
     { printf( "%d\n", i*i+((3*i)-10/i) );
     }
   }
}
```

- Positionieren Sie nun den Cursor auf die öffnende geschweifte Klammer in der zweiten Zeile.
- Nun wählen Sie **„Strukturkontrolle"** aus dem „Suchen"-Menü. Der Cursor wird automatisch auf die korrespondierende schließende geschweifte Klammer in der letzten Zeile positioniert.
- Wenn Sie erneut **„Strukturkontrolle"** wählen, dann wird der Cursor wieder auf die öffnende geschweifte Klammer in der zweiten Zeile gesetzt.

Die Funktion **„Strukturkontrolle"** versucht immer den Cursor auf die zur Klammer unter dem Cursor korrespondierende öffnende oder schließende Klammer zu setzen. Sollte dies nicht möglich sein, so meldet Edward **„Structures do not match!"**. Somit können Sie bei stark verschachtelten Funktionsblöcken in Ihren Programmen auf einfache Weise überprüfen, ob Sie zu jeder öffnenden auch eine schließende Klammer gesetzt haben und Sie müssen nicht erst den Compiler bemühen, um solche leicht zu vermeidenden Syntax-Fehler zu entdecken.

Mit den runden Klammern in der sechsten Zeile des obigen Beispiels können Sie die Funktion „Strukturkontrolle" ausprobieren. Nehmen Sie ruhig einmal eine Klammer heraus und stellen Sie fest, wie Edward darauf reagiert.

3.6.4 Compiler Error-Datei interpretieren, Fehler anspringen

Eine Besonderheit von Edward ist, daß er die Fehlerdateien bestimmter Compiler auswerten kann. Sie müssen nur von Ihrem Compiler eine Fehlerdatei erzeugen lassen und diese mit Edward laden. Die dafür nötigen Funktionen können Sie zum Beispiel auf eine Taste oder ein Menü legen. Bei einem C-Compiler können Sie durch einfachen Aufruf von „CC >ErrorFile ..." die Fehlermeldungen in die Datei „ErrorFile" umleiten. Wenn Sie die Fehlerdatei geladen haben, so positionieren Sie den Cursor auf eine Zeile in der die Zeilennummer eines Fehlers steht. Danach wählen Sie **„Gehe zu Fehler"**. Daraufhin überprüft Edward, ob die Datei, zu der die Fehlerbeschreibung gehört, sich schon im Speicher befindet. Ist dies nicht der Fall, so wird die entsprechende Datei nachgeladen. Daraufhin wird der Cursor auf die Zeile positioniert, in welcher sich der angebene Fehler befindet.

☛ Wenn Sie in einer geladenen Quelldatei einige Veränderungen vornehmen und dabei Zeilen einfügen oder löschen, so kann **„Gehe zu Fehler"** natürlich nicht mehr auf die richtige Zeile positionieren. Sie sollten daher immer die ungefähren Verschiebungen von Zeilen im Kopf behalten.

3.6.5 Auf bestimmte Zeilen positionieren

Mit **„Gehe zur Zeile.."** öffnen Sie ein kleines Formular, in dem sofort ein Feld zur Eingabe einer Zahl bereit ist. Geben Sie die Nummer der gewünschten Zeile ein und drücken Sie <RETURN>. Daraufhin versucht Edward den Cursor auf die gewünschte Zeile zu positionieren.

Mit **„Gehe zum Anfang"** können Sie an den Anfang eines Textes mit **„Gehe zum Ende"** an das Ende eines Textes gelangen.

3.7 Strukturieren von Texten, Lesezeichen

Programme bestehen normalerweise aus Prozeduren, die in weitere kleine Einheiten strukturiert sind (oder zumindest sein sollten). Um ein Programm etwas übersichtlicher darstellen zu können, bietet Ihnen Edward das Konzept der Falten an. Desweiteren können Sie bestimmte Zeilen markieren. Mit den Cursor-Tasten lassen sich so bestimmte Strukturen einer Datei leicht anwählen. Ausserdem können wichtige Zeilen gemerkt und mit einem Tastendruck direkt angesprungen werden (Lesezeichen).

Hinweis

Die Strukturierung eines Textes wird in der zum Text gehörigen Pictogramm-Datei abgespeichert. Näheres hierzu erfahren Sie im Kapitel 3.15 „Globale Einstellungen".

3.7.1 Erzeuge Falte, Zeige Falte, Entferne Falte, Zeige alle Falten, Schließe alle Falten

Hinter dem Konzept der Falten verbirgt sich eine sehr leistungsfähige Methode zur Strukturierung von Programm-Quelldateien.

- Geben Sie nun folgendes kleine Programm ein oder laden Sie die Quelldatei eines von Ihnen erstellten Programms ein.

```
Program Integration(input,output);
{  Demonstriert Prozedur- und Funktionsparameter sowie Stringtypen }
{                                                                  }
{  Es werden die Quadratfunktion nach dem Trapezsummen- und die    }
{  Sinusfunktion nach dem Simpsonverfahren in wählbaren Grenzen    }
{  integriert.                                                     }

Type string20=array[1..20] of char;

Function g(x:real):real;
       { "Sqr" kann nicht direkt als Parameter übergeben werden.
       Grund: Der Ergebnistyp ist nicht definiert, sondern hängt vom
       Parametertyp ab (integer oder real). Dies gilt auch z. B. für
       "succ" und "pred". Solche speziellen Funktionen sind nicht als
       aktuelle Parameter erlaubt.                                 }
  Begin
```

```
    g:=sqr(x)
  End;

Procedure trap(Function f(x:real):real;a,b:real; n:integer; Var
erg:real);
Var s,h:real; i:integer;
  Begin
    h:=(b-a)/n;
    s:=0;
    For i:=1 to n-1 do
      s:=s+f(a+i*h);
    erg:=(0.5*(f(a)+f(b))+s)/n
  End;

Procedure simp(Function f(x:real):real; a,b:real; n:integer; Var
erg:real);
Var t1,t2:real;
  Begin
    trap(f,a,b,n,  t1); { Beachte: eine Function, die "simp" als
                          Parameter }
    trap(f,a,b,2*n,t2); { übergeben wurde, wird hier an "trap"
                          weitergereicht. }
    erg:=(4*t2-t1)/3
  End;

Procedure Tabelle
  ( funk, verf:           string20;
    Function f(y:real):   real;
    Procedure int
    ( function f(y:real):real;
      x1,x2:              real;
      a:                  integer;
      Var                 r:real
    )
  );
{ Man beachte die tolle Schachtelung: Der Prozedurparameter "int" hat
selbst wieder einen Funktionsparameter! }

  Var lo,hi,e:real; i,max:1..100;
  Begin
    writeln("Integral der Funktion f(x)=",funk);
    writeln;
    write('Grenzen: ');readln(lo,hi);
    write('Anzahl:  ');readln(max);
    page;
    writeln('Integration von f(x)=',funk,' durch ',verf,'verfahren');
    writeln;
    For i:=1 to max do
      Begin
        int(f,lo,hi,i,e);
        writeln(i:8,'    ',e)
      End;
```

```
    writeln
  End;
Begin  { Hauptprogramm }
  Tabelle('x^2',   'Trapez' ,g,   trap);
  Tabelle("sin(x)","Simpson", sin,simp)
End.
```

- Bewegen Sie den Cursor auf die erste Zeile der ersten Funktion. Wählen Sie „Markieren" aus dem „Block"-Menü. Positionieren Sie den Cursor nun unter die letzte Zeile der Funktion.
- Jetzt wählen Sie **„Erzeuge Falte"** aus dem „Falte"-Menü. Alle markierten Zeilen bis auf die erste verschwinden nun. Vor dieser erscheint das Zeichen „«". Der Cursor wurde ebenfalls auf diese Zeile bewegt.

Damit haben Sie die erste Falte erzeugt. Alles, was vorher markiert war, steckt nun in der Falte. Nur die erste Zeile ist noch sichtbar. Ein besonderes Zeichen zeigt Ihnen den Beginn der Falte an.

- Wählen Sie nun **„Zeige Falte"**. Daraufhin wird der Inhalt der Falte wieder sichtbar. Die Zeilen innerhalb der Falte beginnen mit dem Zeichen „|". Die letzte Zeile einer Falte wird durch das Zeichen „»" kenntlich gemacht.
- Wählen Sie erneut **„Erzeuge Falte"**, dann wird die Falte wieder „eingeklappt" und die vorher angezeigten Zeilen verschwinden.
- Wenn Sie **„Entferne Falte"** anwählen, so verschwindet die Markierung der gefalteten ersten Zeile, und alle „innereren" Zeilen der Falte erscheinen wieder. Diesmal allerdings unmarkiert. Die Faltung wurde vollständig rückgängig gemacht.

☛ **„Zeige Falte"** und **„Entferne Falte"** arbeiten nur, wenn der Cursor auf die erste Zeile einer Falte positioniert wurde. **„Erzeuge Falte"** faltet einen markierten Block ein oder schließt eine mit **„Zeige Falte"** geöffnete Falte wieder.

- Falten Sie nun alle Prozeduren aus unserem Beispiel ein.
- Jetzt wählen Sie **„Zeige alle Falten"**. Daraufhin erscheinen beide gefalteten Bereiche so, alls wäre für jeden die Funktion **„Zeige Falte"** aufgerufen worden.
- Durch Wahl von **„Schließe alle Falten"** wird das Ganze wieder rückgängig gemacht und nur die erste Zeile jeder Falte ist noch sichtbar.

Mit **„Entferne alle Falten"** entfernen Sie alle Falten aus Ihrem Text.

Selbstverständlich lassen sich die Falten ineinander schachteln. Das bedeutet, das Sie mehrere gefaltete Bereiche in einer übergreifenden Falte zusammenfassen können.

Mit **„Zeige innere Falten"** können Sie auf einen Schlag eine Falte mit all ihren inneren Falten öffnen. **„Schließe innere Falten"** macht dies rückgängig.

Mit **„Entferne innere Falten"** entfernen Sie eine Faltung inklusive all ihrer Unterfaltungen.

3.7.2 Auszeichnen von Zeilen

Bestimmte Zeilen eines Dokuments oder einer Quelldatei, die Sie mit Edward bearbeiten, können Sie als besonders wichtig kennzeichnen. Diese Zeilen können auf besonders einfache Weise angesprungen werden.

Eine Zeile wird durch die Tastenkombination <ALT>+<SHIFT>+<RETURN> ausgezeichnet. Die Zeile wird daraufhin mit einem führenden Ausrufungszeichen dargestellt. Wiederholen Sie diese Tastenkombination, so wird die Auszeichnung zurückgenommen, und das Ausrufungszeichen verschwindet.

3.7.3 Cursor-Bewegung

Mit zwei Tastenkombination können Sie gefalteten Zeilen sehr einfach anspringen. Mit <CTRL>+<CURSOR_UP> wählen Sie eine vor der aktuellen Zeile liegende Zeile an, die einen Faltenstart oder ein Faltenende darstellt. Ebenso können Sie mit <CTRL>+<CURSOR_DOWN> eine nach der aktuellen Zeile liegende Zeile anspringen.

Mit <ALT>+<SHIFT>+<CURSOR_UP> bzw. <ALT>+<SHIFT>+<CURSOR_DOWN> springen Sie eine vor bzw. nach der aktuellen Zeile liegende Zeile an, die Sie ausgezeichnet hatten.

3.7.4 Lesezeichen (Bookmarks)

Beim Arbeiten mit einem Text kommt es oft vor, daß man schnell verschiedene, weit auseinanderliegende Positionen im Text anspringen möchte, an denen man gleichzeitig arbeitet (z.B. ich an diesem Handbuch).

Eine Möglichkeit um dies zu erreichen wäre, die entsprechenden Zeilen auszuzeichnen und dann mit den Cursor-Tasten anzuspringen, dies hat jedoch den Nachteil, daß man unter Umständen mehrere Bewegungen im Text machen muß, um die gewünschte Stelle zu erreichen.

In diesem Fall gibt es aber eine weitere Besonderheit von Edward, die sogenannten „Lesezeichen". Mit der linken <ALT>-Taste und einer Zahlentaste des alphanumerischen Teils der Tastatur können Sie eine Zeile zwischenmerken. Wenn Sie nun an eine andere Stelle des Textes fahren und die linke <AMIGA>-Taste in Verbindung mit derselben Zahlentaste drücken, so springen Sie augenblicklich an die alte Stelle im Text.

Mit der Standard-Tastaturbelegung Edwards können Sie zehn Lesezeichen benutzen, theoretisch erlaubt Ihnen Edward jedoch 65536 Lesezeichen pro Text.

- Laden Sie eine Datei ein.
- Drücken Sie die linke <ALT>-Taste und die **„1"**.

Es erscheint in der Titelzeile des Fensters die Meldung **„Markierung gesetzt"**.

- Fahren Sie an eine beliebige andere Stelle des Textes.

- Drücken Sie nun die linke <AMIGA>-Taste und wieder die **„1"**. Und schwupps sind Sie wieder an der alten Position im Text.

3.8 Das Edit-Prefs Formular

Mit dem „Edit Prefs"-Formular, das Sie durch Wahl von „Edit Prefs..." aus dem „Text"-Menü erhalten, können Sie für jeden Text den Sie mit Edward bearbeiten diverse Einstellungen vornehmen.

Auto-Indent

Wenn Sie Auto-Indent aktivieren, wird bei der Standard-Tastaturbelegung auch nach Drücken der <RETURN>-Taste die neue Zeile entsprechend der alten eingerückt, wie es nach Drücken der <ENTER>-Taste der Fall wäre.

☞ Siehe auch Kapitel 3.2 „Eingabe und Bearbeitung von Texten"

Word Wrap

Durch Klick auf diesen Schalter wird der „Word-Wrap"-Modus aktiviert.

☞ Über den „Word-Wrap"-Modus erfahren Sie mehr im Kapitel 3.9. „Word-Wrap".

Überschreiben

Ist Überschreiben angewählt, dann überschreiben Sie jeweils das Zeichen auf dem der Cursor steht durch das neugetippte. Normalerweise würde Edward das Zeichen einfügen, Überschreiben überschreibt dagegen die alten Zeichen.

TAB in Spaces

Ist diese Option aktiv, so werden alle eingegeben TABs in einer Zeile automatisch in entsprechend viele Leerzeichen umgewandelt.

Zeige TABS
Zeige Spaces

Mit diesen drei Symbolen können Sie angeben, ob Tabs, Spaces und Zeilenenden einfach durch Leerräume dargestellt werden sollen (Normaleinstellung) oder besonders gekennzeichnet dargestellt werden sollen.

Übersetze CRs

Dateien von anderen Computersystemen enthalten unter Umständen ein zusätzliches Zeichen um ein Zeilenende anzuzeigen. Durch Aktivieren dieser Option wird ein solches Zeichen automatisch beim Laden des Textes entfernt.

TAB Weite

Tab Weite gibt die Anzahl von Zeichen zwischen zwei Tabularpositionen an. Sie können diesen Abstand im Bereich 1 bis 20 wählen.

Rechter Rand

Mit Rechter Rand geben Sie die Position in einer Zeile an, ab der im „Word-Wrap"-Modus umgebrochen werden soll.

Klammern Einzug

Der hier eingetragene Wert gibt die Anzahl TABS an, die bei einem automatischen Einzug gesetzt werden.

OK

Wählen Sie USE, so werden die von Ihnen gewählten Einstellungen für den Text übernommen. Falls Sie „Rechter Rand" geändert haben, so wird der komplette Text dementsprechend neu umformatiert und der Cursor danach auf die erste Zeile des Textes gesetzt.

Abbruch

Verwirft die von Ihnen vorgenommenen Einstellungen.

3.9 „Word-Wrap"

Zum Erstellen von Fließtexten, wie sie unter anderem für DTP-Programme benötigt werden, können Sie Edward in dem „Word-Wrap"-Modus versetzen. „Word-Wrap" bezeichnet ein spezielles Verhalten des Editors, bei dem eingegebene Zeilen beim Überschreiten einer bestimmten Länge umgebrochen werden. Ein Wort, das mitten auf der Umbruchposition liegt, wird automatisch in die nächste Zeile gezogen. Wird diese Zeile dadurch ebenfalls zu lang, so wird das Ganze fortgesetzt und zwar bis an das Ende eines Absatzes.

Ein Absatz wird dadurch gekennzeichnet, daß Sie nach der letzten Zeichenfolge des Absatzes <RETURN> drücken. Im Gegensatz zum normalen Editier-Modus von Edward ist es im „Word-Wrap"-Modus nicht notwendig, die <RETURN>-Taste zu drücken, um eine neue Zeile zu erhalten.

Vielmehr erzeugt Edward automatisch beim Umbruch neue Zeilen. Sie brauchen also nur drauflos-zutippen. Daher stammt auch die Bezeichnung „Fließtext".

☞ Die Umbruchposition wird durch Angabe des rechten Randes im „Edit Prefs"-Formular angegeben. Der „Word-Wrap"-Modus kann ebenfalls durch dieses Formular ein- und ausgeschaltet werden. Wie Sie dabei vorgehen müssen, haben Sie im Kapitel 3.8. „Das Edit-Prefs Formular" gelesen.

Im Gegensatz zu anderen Editoren, die einen Absatz erst durch Wahl eines bestimmten Menüpunktes einmalig umbrechen können, formatiert Edward den Text schon während der Eingabe. Auch beim Löschen innerhalb eines Absatzes werden Worte aus folgenden Zeilen eines Absatzes wieder nachgezogen, wenn in der aktuellen Zeile durch das Löschen Platz für sie gewonnen wurde.

Neben dem automatischen Umbruch des „Word-Wrap"-Modus können Sie einen Text auch nachträglich umformatieren.

Hierzu wählen Sie entweder **„Format Paragraph"**, um einen Absatz neu umzubrechen oder **„Format Text"**, um den gesamten Text neu umzubrechen. Beide Menüpunkte befinden sich im „Extras"-Menü.

3.10 Makros

Eine Besonderheit von Edward, die Ihnen viel Arbeit erleichtern kann, ergibt sich aus der Möglichkeit, Makros aufzeichnen und abspielen zu können.

Fast alle Aktionen, die Sie beim Editieren eines Textes ausführen, können Sie wie mit einem Kassettenrekorder aufnehmen und bei Bedarf wiedergeben lassen. Das bedeutet, daß Sie sich alle gleichförmigen langweiligen Vorgänge, die Sie oft wiederholen müßten, durch Makros abnehmen lassen können.

3.10.1 Makros aufzeichnen & abspielen

Wenn Sie ein Makro aufzeichnen wollen, so wählen Sie den Menüpunkt **„Aufnehmen"** aus dem „Macro"-Menü. Daraufhin werden alle folgenden Aktionen, die Sie vornehmen, aufgezeichnet. Mit **„Anhalten"** aus dem „Macro"-Menü beenden Sie eine Aufzeichnung. **„Abspielen"** spielt das Makro dann für Sie ab.

- Laden Sie einen größeren Text ein.

Angenommen Sie wollten nun zu Beginn jeder Zeile die Zeichenfolge **„Start>"** und am Ende jeder Zeile die Zeichenfolge **„<End"** einfügen. Normalerweise müßten Sie jedesmal an den Anfang einer Zeile gehen, **„Start>"** eintippen, dann ans Ende einer Zeile springen, **„<End"** eingeben und in die nächste Zeile gehen. Wenn Sie die Möglichkeiten der Makros ausnutzen, wird das Ganze für Sie sehr viel einfacher.

- Fahren Sie mit dem Cursor in die erste Zeile Ihres Textes.
- Wählen Sie nun den Menüpunkt **„Aufzeichnen"** aus dem „Macro"-Menü.

Alle folgenden Aktionen werden nun aufgezeichnet.

- Drücken Sie <SHIFT> + <CursorLeft>. Damit springen Sie an den Anfang einer Zeile.
- Tippen Sie nun **„Start >"** ein.
- Anschliessend bewegen Sie den Cursor mit <SHIFT> + <CursorRight> an das Ende der aktuellen Zeile.
- Geben Sie nun **„< End"** ein.
- Fahren Sie mit dem Cursor eine Zeile tiefer.
- Wählen Sie **„Anhalten"** aus dem „Macro"-Menü. Damit haben Sie die Aufzeichnung beendet. Alle Aktionen zwischen **„Aufnehmen"** und **„Anhalten"** wurden aufgezeichnet und stehen nun zu Ihrer Verfügung.

Sie können nun jederzeit das gerade aufgezeichnete Makro abspielen.

- Wählen Sie **„Abspielen"** aus dem „Macro"-Menü. Sie sehen, wie die folgende Zeile in Sekundenbruchteilen ebenfalls mit **„Start >"** und **„< End"** eingerahmt wird. Sie können diesen Vorgang nun beliebig oft wiederholen und sich damit viel Tipparbeit ersparen.

Statt obigem zugegebenermaßen etwas sinnlosem Beispiel werden Ihnen in der Praxis sicherlich sehr viel nützlichere Einsatzmöglichkeiten für die Makrofunktionen einfallen.

3.10.2 Makros speichern & abspielen

Die mit **„Aufnehmen"** aufgezeichneten Makros stehen hier nur kurze Zeit zur Verfügung. Wenn Sie ein neues Makro aufzeichnen oder Edward verlassen, so ist Ihr zuletzt aufgezeichnetes Makro für immer verloren. Damit Sie ein besonders praktisches Makro nicht immer wieder neu aufzeichnen müssen, bietet Ihnen Edward die Möglichkeit ein Makro abzuspeichern und abgespeicherte Makros erneut abzuspielen.

- Falls Sie das Beispiel aus Abschnitt 1) noch nicht durchgespielt haben, so sollten Sie es jetzt tun. Wir benötigen für die folgenden Schritte nämlich ein frisch aufgezeichnetes Makro.
- Wählen Sie nun den Menüpunkt **„Speichern als ..."** aus dem „Makro"-Menü. Es erscheint der Dateiauswahlfenster. Speichern Sie das Makro nun unter irgendeinem Namen ab.
- Zeichnen Sie ein anderes Makro auf oder verlassen Sie Edward und starten ihn danach erneut. Dies dient dazu das aktuelle Makro zu „vergessen". Am einfachsten ist dies für Sie, wenn Sie irgendwelche neuen Aktionen wie in Abschnitt 1) beschrieben aufnehmen.
- Wählen Sie nun **„Öffnen und abspielen"** aus dem „Makros"-Menü. Mit dem Dateiauswahlfenster geben Sie nun das im vorletzten Schritt abgespeicherte Makro an.

Sie werden sehen, wie Edward automatisch die aktuelle Zeile mit **„Start >“** und **„< End“** umrahmt.

Somit können Sie auch ein gespeichertes Makro beliebig oft wiederaufrufen.

☛ Ein mit **„Öffnen und abspielen“** abgespieltes Makro läßt sich jederzeit mit **„Abspielen“** aus dem „Makro“-Menü aufrufen, ohne erneut geladen werden zu müssen. Ein altes von Ihnen aufgezeichnetes Makro wird durch **„Öffnen und abspielen“** überspielt.

☞ Im Zusammenhang mit „ARexx“ ergeben sich einige weitere Möglichkeiten. Lesen Sie dazu bitte das Kapitel 3.17 „Edward - Kommunikation mit ARexx“.

3.11 Fenster

Sie wissen jetzt schon, wie man verschiedene Texte bearbeitet. Vielleicht haben Sie sich auch schon gefragt, wie man verschiedene Texte oder mehrere Ausschnitte eines bestimmten Textes gleichzeitig anzeigen kann.

Edward ermöglich dies durch die sogenannten Fenster. Was ein Fenster auf der Workbench oder im CLI bedeutet ist klar. Bei Edward ist ein Fenster ein normales Amiga-Fenster verbunden mit den Informationen, die Edward benötigt, um darin einen bestimmten Text oder Textausschnitt anzuzeigen. Wie das genau funktioniert, erkläre ich Ihnen in den folgenden Abschnitten.

3.11.1 Erzeugen von Fenstern

Nach dem Start öffnet Edward sofort ein Fenster, in welchem Sie einen Text bearbeiten können.

- Geben Sie irgendeinen Text ein.
- Wählen Sie aus dem „Fenster“-Menü den Eintrag **„Neu“**. Sie sehen kurz, wie sich ein Fenster öffnet. Dieses Fenster liegt genau über dem ersten Fenster von Edward und hat auch denselben Inhalt. Wenn Sie nur flüchtig zugeschaut haben, werden Sie keinen Unterschied in der Anzeige bemerkt haben.
- In der rechten unteren Ecke befindet sich ein Symbol mit welchem Sie die Größe des Fenster verändern können. Halbieren Sie das Fenster in der Höhe und der Breite. Sie sehen nun, daß darunter Ihr altes Fenster liegt.
- Verschieben Sie das Fenster in die rechte, untere Ecke.
- Wählen Sie den Eintrag **„Neu“** aus dem „Text“-Menü.
- Laden Sie einen Text mit **„Projekt/Öffnen“**, oder geben Sie einen weiteren Text ein.

Für den weiteren Verlauf unserer Übung sollten Sie jetzt noch ein Fenster öffnen, es verkleinern und in die linken unteren Ecke verschieben. Danach laden Sie einen weiteren Text oder Sie geben einfach ein paar Zeilen ein.

3.11.2 Wahl eines bestimmten Fensters

Es gibt verschiedene Möglichkeiten, ein bestimmtes Fenster zu aktivieren.

- Am einfachsten fahren Sie mit dem Mauspfeil im entsprechenden Fenster an die Position im Text, die Sie bearbeiten wollen und drücken die linke Maustaste. Das Fenster wird daraufhin aktiviert und der Cursor erscheint unter dem Mauspfeil. Über das Tiefen-Gadget (bzw. bei Kickstart 1.3 die beiden Tiefengadgets) können Sie an jedes Fenster herankommen und es dann wie oben beschrieben mit dem Mauspfeil aktivieren.
- Wählen Sie auf diese Weise das Fenster in der rechten unteren Ecke an.

Ebenso wie die Texte werden auch die Fenster in einer Liste verwaltet. Auf einfache Weise können Sie das Fenster an der Listenposition vor oder auch nach dem momentan aktuellen wählen. Hierbei gilt ebenfalls alles zu den Texten gesagte. Das momentan aktuelle Fenster sollte das von Ihnen erzeugte in der linken unteren Ecke sein.

- Wählen Sie aus dem „Fenster"-Menü den Eintrag **„Vorheriges"**. Damit haben Sie wieder das Fenster in der rechten, unteren Ecke aktiviert.
- Wählen Sie erneut **„Vorheriges"**. Die beiden kleinen Fenster sind nun plötzlich verschwunden. Einzig das grosse Fenster, das beim Start von Edward automatisch geöffnet wurde, ist nun sichtbar. Das liegt daran, daß bei einer Aktivierung eines Fensters durch **„Vorheriges"** oder **„Nächstes"** dieses immer automatisch in der Vordergrund geholt wird.

3.11.3 Schließen von Fenstern

Sie können ein Fenster auf zwei Arten schließen. Entweder Sie bewegen dem Mauspfeil auf das Schließen-Symbol in der linken, oberen Ecke des Fensters und drücken die linke Maustaste, oder Sie wählen den Eintrag **„Schließen"** aus dem „Fenster"-Menü, der das momentan aktive Fenster schließt.

- Aktivieren Sie, wie im vorigen Abschnitt beschrieben, das große Fenster.
- Schließen Sie es nun durch den **„Schließen"**-Menüeintrag. Nun sind nur noch die beiden kleinen Fenster vorhanden.
- Schließen Sie nun das Fenster in der rechten, unteren Ecke mit dem Mauspfeil.

Wenn Sie nun mehrfach **„Nächster"** oder **„Vorheriger"** aus dem „Text"-Menü (nicht dem „Fenster"-Menü !) wählen, dann stellen Sie fest, daß die drei verschiedenen Texte alle noch da sind, obwohl Sie zwei Fenster geschlossen haben. Das liegt daran, daß beim Schließen eines Fensters nur die Anzeige eines Textes beendet wird. Der Text wird dadurch aber nicht freigegeben.

Wichtig

Wenn Sie versuchen, das letzte Fenster von Edward zu schließen, geht Edward davon aus, daß Sie die Arbeit beenden wollen. Sollten Sie noch Texte haben, deren Änderungen nicht abgespeichert wurden, so werden Sie durch ein Dialog-Fenster darauf aufmerksam gemacht. Gegebenenfalls können Sie die Operation mit „Nein" abbrechen. Wählen Sie „Ja", dann gehen alle vorher gemachten Änderungen verloren.

- Lesen Sie hierzu auch Kapitel 3.3.3 „Freigeben eines Textes".

3.11.4 Arbeiten mit mehreren Ansichten eines Textes

Mit Edward können Sie sich mehrere Ansichten desselben Textes anzeigen lassen und damit arbeiten.

- Sie sollten nun nur noch ein Fenster von Edward geöffnet haben. Sollte dies nicht der Fall sein, so sorgen Sie bitte mit den oben beschriebenen Schritten dafür.
- Bewegen Sie nun das Fenster in die linke, obere Ecke des Bildschirms und verändern Sie seine Größe so, daß es die gesamte Breite und etwa ein Drittel der Höhe des Bildschirms ausfüllt.

Damit Sie die in diesem Abschnitt beschriebenen Möglichkeiten richtig kennenlernen können, sollten Sie nun einen Text einladen oder eingeben, der etwa 100 Zeilen oder mehr umfasst.

- Erzeugen Sie zwei weitere Fenster mit denselben oben beschriebenen Ausmassen und ordnen Sie diese untereinander so an, daß es keine Überlappungen zwischen ihnen gibt.
- Aktivieren Sie nun das mittlere Fenster und fahren Sie mit dem Cursor nach unten in die 40zigste Zeile.
- Aktivieren Sie das unterste Fenster und fahren Sie mit dem Cursor in die letzte Zeile des Textes.

Sie können nun in jeder der drei Ansichten des Textes an verschiedenen Positionen weiterarbeiten. Der Vorteil ist, daß Sie in den beiden anderen Fenstern eine Übersicht über ganz andere Textstellen haben, als jene, die Sie gerade bearbeiten. Das kann besonders beim Erstellen von Quellcode für eine bestimmte Programmiersprache von Vorteil sein.

Hinweis

Alle Änderungen, die Sie an einem Text vornehmen, inklusive der Löschung desselben, werden nur in dem aktuellen Fenster auch angezeigt. Erst wenn Sie eine andere Ansicht des Textes in einem anderen Fenster anwählen, wird auch dort die Anzeige aktualisiert. Lassen Sie sich davon nicht verwirren.

Um Ihnen das Arbeiten mit verschiedenen Ansichten eines Textes zu erleichtern, bietet Edward zusätzliche Funktionen an.

- Schließen Sie alle Fenster bis auf eines.
- Wählen Sie nun **„volle Größe"** aus dem „Fenster"-Menü.

Sie sehen, daß das letzte Fenster sich wieder auf die maximale Größe ausdehnt. So erhalten Sie schnell wieder den Überblick, falls ein Fenster zu klein war oder Sie durch Schließen anderer Fenster mehr Platz gewonnen haben.

- Wählen Sie nun **„horizontal teilen"** aus dem „Fenster"-Menü. Daraufhin teilt sich das aktuelle Fenster horizontal in zwei gleichgroße Fenster auf. So erhalten Sie schnell die nötigen Fenster für verschiedene Ansichten desselben Textes. Mit **„vertikal teilen"** unterteilen Sie genauso ein Fenster vertikal in zwei gleichgroße Fenster.

3.12 Verlassen des Editors

3.12.1. Unterbrechen der Arbeit

Edward bietet Ihnen besonderen Komfort, falls Sie die aktuelle Arbeit an einem oder mehreren Texten unterbrechen und den Editor verlassen wollen. Wenn Sie im „Projekt"-Menü den Eintrag **„Projekt & Ende"** wählen, so speichert Edward zusätzliche Informationen über alle momentan verwendeten Fenster und Texte ab. Wenn Sie Edward erneut starten, dann öffnet er automatisch Fenster und lädt Texte ein, so daß Sie die Arbeit in derselben Umgebung fortsetzen können, in der Sie sie unterbrochen haben.

Wichtig

Falls Sie Änderungen an Texten vornahmen, ohne diese zu speichern und daraufhin **„Projekt & Ende"** gewählt haben, so werden Sie über ein Dialog-Fenster gefragt, ob sie diese Texte „vergessen" wollen. Antworten Sie hier mit „Nein", dann wird die Operation abgebrochen. Wählen Sie „Ja", dann gehen die letzten Änderungen verloren!

- Damit Sie einen kleinen Eindruck über die Möglichkeiten der Arbeitsunterbrechung erhalten, sollten Sie nun Fenster mit verschiedenen Texten erzeugen.
- Wählen Sie nun den Menüpunkt **„Projekt & Ende"** aus dem „Projekt"-Menü. Sie befinden sich nun wieder auf der Workbench oder im CLI, je nachdem, wie Sie Edward gestartet haben.
- Starten Sie Edward erneut. Automatisch öffnen sich die Fenster wieder so, wie Sie vor dem Verlassen angeordnet waren und zeigen die zuletzt bearbeiteten Texte an. Sie können nun einfach mit der Arbeit fortfahren.

3.12.2 Verlassen des Editors

Im Gegensatz zu **„Projekt & Ende"** verlassen Sie mit **„Ende"** aus dem „Projekt"-Menü den Editor, ohne daß Informationen zur momentanen Arbeitsumgebung gespeichert werden.

Wichtig

Falls Sie Änderungen an Texten vornahmen, ohne diese zu speichern und daraufhin **„Ende"** gewählt haben, so werden Sie über ein Dialog-Fenster gefragt, ob sie diese Texte „vergessen" wollen. Antworten Sie hier mit „Nein", dann wird die Operation abgebrochen. Wählen Sie „Ja", gehen die letzten Änderungen verloren!

3.13 Iconify & HotKey

3.13.1 Iconify

Benutzen Sie Edward auf dem Workbench-Screen, so wird es manchmal sehr lästig sein, wenn Edward mit seinen Fenstern etwas im Hintergrund verdeckt. Vielleicht wollen Sie einfach ein Programm von der Workbench aus starten oder in der Shell den Compiler anwerfen. Damit Sie sich nicht durch alle Fenster Edwards durchklicken müssen, gibt es die Funktion Ikonifizieren

Wählen Sie einfach aus dem „Fenster"-Menü den Eintrag **„Ikonifizieren"**. Schon verschwinden alle Fenster, und es erscheint links oben auf dem Bildschirm ein Fenster, das nur aus einer Titelzeile besteht. In dieser steht **„Iconified Edward"**. Edward ist nun nicht mehr aktiv. Das kleine Fenster könnnen Sie irgendwohin schieben, wo es Sie nicht stören kann.

Wollen Sie Edward wieder aktivieren, so haben Sie zwei Möglichkeiten. Die zweite wird im Abschnitt HotKey beschrieben.

Die erste Möglichkeit besteht darin, das Fenster mit dem Titel **„Iconified Edward"** zu aktivieren und dann die rechte Maustaste zu drücken. Daraufhin baut Edward wieder alle seine Fenster auf und Sie können weiterarbeiten.

3.13.2 HotKey

Um Edward nach einem Iconify wieder aktivieren zu können, oder um einfach das Fenster mit dem zuletzt bearbeiteten Text in den Vordergrund zu bekommen, gibt es den HotKey.

Der HotKey ist eine Taste in Verbindung mit bestimmten Qualifiern (SHIFT, ALT, CTRL, LEFT_AMIGA). Wird diese Taste gedrückt, so wird das Fenster mit dem zuletzt bearbeiteten Text und der Bildschirm auf dem sich dieses Fenster befindet in den Vordergrund gebracht. War Edward im Iconify-Zustand, so wird dieser aufgehoben, und alle Fenster werden wieder geöffnet.

☞ Näheres zur Definition des HotKeys erfahren Sie im Kapitel 3.16 „Benutzerdefinierte Menüs, Tastaturbelegung und HotKey".

4.14 Der Kommandozeilen-Interpreter

Die Bedienung Edwards über Menüs und Formulare stellt für den Anfänger sicherlich eine große Erleichterung dar, der Profi ist in vielen Fällen jedoch bereit auf diesen Komfort zu verzichten, wenn er nur besonders schnell die wichtigsten Funktionen benutzen kann.

Dies kann zum einen durch Definition von „Short-Cuts" wichtiger Menüpunkte und durch Belegung von Funktionstasten etc. geschehen. Eine weitere mächtige Möglichkeit ist der Kommandozeilen-Interpreter Edwards. Hier können viele Befehle und Befehlskombinationen Edwards direkt eingegeben und ausgeführt werden.

Die Eingabe einer Kommandozeile leiten Sie durch die <ESC>-Taste ein. Daraufhin erscheint der Cursor in der untersten Zeile des aktiven Textfensters. Die dort angezeigte Textzeile wird durch eine Leerzeile ersetzt, in der Sie nun Ihre Eingaben machen können. Ein erneutes <ESC> bricht die Eingabe ab. Drücken Sie stattdessen <RETURN> oder <ENTER> werden die von Ihnen eingegebenen Kommandos ausgewertet.

Mit <CTRL>+<O> gelangen Sie ebenfalls in den Eingabemodus des Kommando-Interpreters mit der Besonderheit, daß Ihre letzte Eingabe wieder zur Verfügung steht und geändert werden kann.

<CTRL>+<G> bewirkt eine erneute Ausführung der zuletzt eingebenen Kommandos.

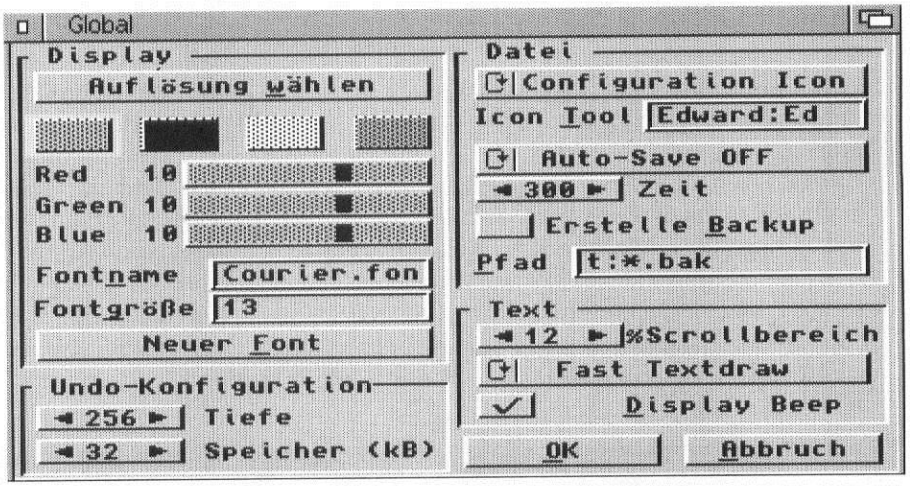
☞ Weitere Informationen über den syntaktisch korrekten Aufbau einer Kommandofolge erhalten Sie in Kapitel 3.16 „Benutzerdefinierte Menüs, Tastaturbelegung und HotKey".

☞ Alle verfügbaren Kommandos Edwards mit Kurzerläuterungen und Parameterbeschreibung sind im Anhang enthalten.

3.15 Globale Einstellungen

3.15.1 Das „Globale Einstellungen"-Formular

Mit dem „Globale Einstellungen"-Formular, das sich öffnet, wenn Sie **„Einstellungen..."** aus dem „Projekt"-Menü wählen, können Sie verschiedene Einstellungen verändern, die den gesamten Editor betreffen.



Auflösung wählen

Bei Klick auf dieses Symbol erscheint ein Dialog-Fenster aus dem Sie die gewünschte Auflösung auswählen können.

Farben

Mit den folgenden Symbolen können Sie die vier Farben einstellen, in denen die Fenster und Texte Edwards auf einem eigenen Bildschirm dargestellt werden sollen. Diese Felder sind nur anwählbar, wenn Sie Edward auf einem eigenen Screen betreiben! Auf der Workbench steht Ihnen zur Veränderung der Farben der Prefs-Einsteller zur Verfügung.

Fontname

In dieses Texteingabefeld können Sie den Namen des Zeichensatzes angeben, in welchem die Texte Edwards dargestellt werden sollen.

Fontgröße

Dieses Texteingabefeld erfaßt die Fontgröße.

Wichtig

Für den beschleunigten Bildaufbau bei der Textdarstellung mußten bei der Entwicklung Edwards gewisse Kompromisse eingegangen werden. Daher ist es leider nicht möglich Proportional-Zeichensätze zu wählen. Sollten Sie einen falschen Zeichensatz gewählt haben, so erfolgt keine Änderung der Textdarstellung.

Neuer Font

Durch Anklicken dieses Symbols öffnet sich ein Dialog-Fenster, mit dem Sie auf komfortable Weise den gewünschten Zeichensatz wählen können.

Undo Konfiguration

Mit diesen beiden Zahlenfeldern kontrollieren Sie das Verhalten von Edward im Zusammenhang mit der Undo-Funktion. Mit Undo-Depth geben Sie an, wie viele Operationen, die einen Text verändern, gemerkt werden sollen. Sollte Sie an einem Text eine Anzahl von Änderungen vornehmen, die diesen Wert überschreitet, so werden alte Änderungen „vergessen". Einfache Operationen, wie „Zeichen löschen" oder „Neues Zeichen einfügen" benötigen keinen Zusatzspeicher zum Merken. Das Ausschneiden eines Blockes dagegen kann nur gemerkt werden, wenn genügend Speicherplatz bereitgestellt wurde, um den ausgeschnittenenen Block zusätzlich speichern zu können. Würden Sie nun sehr oft Blöcke ausschneiden und kopieren, so würde sehr viel Speicher in Anspruch genommen. Mit „UndoMem" begrenzen Sie den Speicherplatz, den die Undo-Funktion in Anspruch nimmt, auf einen Maximal-Wert. Wird dieser überschritten, so werden alte Operationen „vergessen", um genügend Speicher wieder freizugeben.

Wichtig

Da die Undo-Funktion für jeden Text die notwendigen Informationen zum Rücknehmen einer Veränderung merken muß, gilt Undo-Mem auch als Wert für jeden Text. Das bedeutet, daß Sie für jeden Text, den Sie bearbeiten, im ungünstigsten Fall die Menge von „UndoMem" zusätzlich an Speicher beanspruchen.

Mit den anschließend beschriebenen Symbolen können Sie das Verhalten Edwards beim Speichern von Dateien angeben.

Datei

Mit diesem Blätter-Symbol geben Sie an, ob Edward beim Speichern einer Datei ein Pictogramm anlegen („Configuration Icon") oder darauf verzichten soll („No Icon").

Wichtig

Nur wenn Sie die Option „Configuration Icon" gewählt haben, werden beim Sichern einer Datei wichtige Zusatzinformationen ebenfalls mitabgespeichert. Dazu gehören unter anderem die aktuelle Position in Ihrem Text, der „Word-Wrap"-Modus und - sehr wichtig - die Strukturierung des Textes durch ausgezeichnete Zeilen.

Configuration Icon

Wenn Sie ein Pictogramm zu jedem Text abspeichern lassen, so können Sie mit diesem Eingabefeld bestimmen, welches Programm beim Doppelklick auf das Pictogramm des gespeicherten Textes geladen werden soll. Im Normalfall sollte dies „Edward:Edward" sein.

Auto-Save

Edward kann die von Ihnen bearbeiteten Texte nach einer bestimmten Zeitspanne abspeichern. Mit dem Blätter-Symbol haben Sie die Wahlmöglichkeit zwischen :

- **„Auto-Save OFF"**, das bedeutet das automatische Abspeichern von Texten ist abgeschaltet.
- **„Auto-Save"**, dann speichert Edward nach der im Zeit-Feld angebenen Zeitspanne alle veränderten Texte automatisch ab.
- **„Confirm Auto-Save"**, dann fragt Edward vor dem automatischen Speichern von Texten um Erlaubnis.

Zeit

Mit diesem Einsteller können Sie die Zeit in Sekunden angeben, die verstreichen darf, bis Edward automatisch die geänderten Texte abspeichert.

Erstelle Backup

Hiermit bestimmen Sie, ob beim Speichern einer Datei eine möglicherweise schon vorhandene Datei gleichen Namens in ein anderes Verzeichnis umkopiert und umbenannt werden soll.

Pfad

In diesem Eingabefeld geben Sie ein spezielles Muster an. Dieses besteht aus dem Pfadnamen gefolgt von einer Zeichenkette, die ein „*" enthalten sein kann. Beim Anlegen eines Backups einer Datei wird diese in den angebenen Pfad umkopiert. Daraufhin wird sie nach der oben erwähnten Zeichenkette umbenannt wobei der „*" durch den alten Namen der Datei ersetzt wird.

Scrollbereich

Mit Scrollbereich ist der Bereich innerhalb eines Textfensters gemeint, in dem der Cursor das Scrollen des Textes bewirkt. Sie können den Abstand des Cursors zu den Fenster-Randkomponenten in Prozent angeben.

Textdraw

Durch das Blätter-Symbol können Sie einstellen, ob die Textausgabe mit Betriebssystem-Funktionen oder direkt gemacht werden soll. Diese Einstellmöglichkeit erhöht die Kompatibilität Edwards zu späteren Betriebssystem-Versionen und zu neuen Grafikkarten mit besonderen Fähigkeiten.

Display Beep

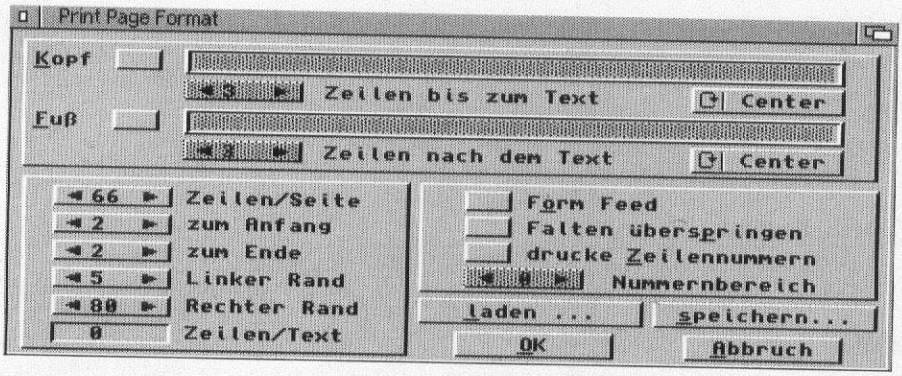
Mit diesem Symbol läßt sich der Display-Beep, der durch jeden Fehler erzeugt wird, abschalten.

OK
Abbruch

Mit diesen beiden Symbolen übernehmen Sie die Einstellungen des Formulars oder Sie brechen den Vorgang ab.

3.15.2 Das „Seitenformat"-Formular

Mit dem „Seitenformat"-Menü können Sie bestimmen, wie ein Text auf dem Drucker ausgegeben werden soll.



Kopf
Fuß

Mit diesen Gadgets geben Sie an ob eine Kopf- bzw. eine Fußzeile ausgedruckt werden soll.

Hier wird der Text für die Kopf- bzw. Fußzeile angegeben. Spezielle Kürzel in diesem Text werden beim Druck interpretiert. Möglich sind hier:
„%D" = Tag, „%M" = Monat, „%Y" = Jahr, „%h" = Stunde, „%m" = Minuten und „%s" = Sekunden. Durch diese Kürzel kann die aktuelle Zeit beim Ausdruck einer Seite in der Kopf- bzw. Fußzeile eingefügt werden.
Weiterhin kann durch „%p" die Seitennummer eingefügt werden.

Zeilen bis zum Text

Gibt die Anzahl Zeilen zwischen der Kopfzeile und dem Text an.

Zeilen nach dem Text

Analog dazu gibt dieser Einsteller die Anzahl Zeilen zwischen Text und Fußzeile an.

Mit den Felder rechts daneben läßt sich die Kopf- bzw. Fußzeile zentriert, links- oder rechtsbündig drucken.

Zeilen/Seite

Hiermit geben Sie an, wieviele Zeilen inklusive Kopf- und Fußzeile, sowie aller Abstände auf ein Blatt passen.

zum Anfang

Hiermit defininieren Sie den Abstand in Zeilen vom Blattbeginn zur ersten Zeile, die gedruckt werden soll (Dies kann die Kopfzeile oder die erste Textzeile sein).

zum Ende

Gibt analog zu „zum Anfang" den Abstand in Zeilen zum Ende des Blattes an.

Linker Rand

Der Abstand des linken Randes in Zeichen.

Rechter Rand

Die Breite einer Zeile in Zeichen.

Zeilen/Text

Dient zur reinen Information. Sie sehen, wieviele Zeilen für die Ausgabe des Textes übrigbleiben.

Form Feed

Nach dem Drucken aller Seiten wird ein zusätzlicher FormFeed gesendet, wenn Sie diese Option aktivieren.

Falten überspringen

Hiermit geben Sie an, ob beim Druck eingefaltete Zeichen übersprungen werden sollen.

drucke Zeilennummern

Wählen Sie diese Option an, so wird jede gedruckte Zeile durchnumeriert.

Nummernbereich

Die Stellenanzahl einer Zeilennummer kann hier angegeben werden.

laden

Laden eines bestimmten Druckformates.

speichern

Speichern der Einstellungen.

OK

Die Einstellungen werden übernommen.

Abbruch

Die Einstellungen werden verworfen.

3.16 Benutzerdefinierte Menüs, Tastenbelegung, HotKey

In den im Verzeichnis EDWARD: abgelegten Dateien mit der Endung „.DEF" befinden sich die Definitionen für die Menüs sowie die Tastaturbelegung Edwards. Diese sind in weiten Grenzen veränderbar. Der Aufbau der Definitionsdatei lehnt sich dabei an eine sehr primitive Programmiersprache an, besitzt also eine ganz einfache Syntax. Die folgenden Abschnitte werden Ihnen das näher erläutern.

3.16.1 Kurze Einführung in die EBNF-Darstellung

Tja, um Ihnen die Syntax der Definitionen ein bißchen näherzubringen muß ich jetzt leider etwas weiter ausholen und die Erweiterte-Backus-Naur-Darstellung einführen. Hinter diesem ominösen Namen, der jedem Informatik-Studenten mehr oder weniger bekannt ist, verbirgt sich eine Darstellungsform für die Syntax von Programmiersprachen. Mächtig kompliziert ist sie deswegen aber nicht.

Eine Definition beginnt immer mit dem Namen des zu Definierenden, daraufhin folgt ein „="-Zeichen. An dieses schließt sich die eigentliche Definition an. Sie gibt im wesentlichen an, welche Zeichenfolgen syntaktisch korrekt sind. Dies geschieht durch direkte Angabe von Zeichenfolgen oder der Angabe einer anderen Definition, in der dann nachgeschaut werden muß. Zeichenfolgen werden immer in Anführungsstrichen wiedergegeben. Jede Definition wird mit einem „." abgeschlossen.

Beispiel 1:

```
Wort = "Mahlzeit" .
```

Würde die Syntax unser Sprache nur aus obiger Definition bestehen, so wäre die einzige syntaktisch korrekte Zeichenfolge „Mahlzeit". Zugegebenermaßen etwas dürftig. Mir sind allerdings Zeitgenossen bekannt, deren Sprachschatz nicht signifikant umfangreicher ist.

Beispiel 2:

```
KlammerAusdruck = "(" Ausdruck ")".
Ausdruck = "Mahlzeit" .
```

In diesem Beispiel nimmt die Definition von Klammerausdruck bezug auf die Definition von Ausdruck. Der einzige korrekte Klammerausdruck ist „(Mahlzeit)".

Um anzuzeigen, daß mehrere Zeichenfolgen einer Definition genügen, dient folgende Notation : Name = (... | ...). wobei „..." für eine Zeichenfolge oder eine andere Definition steht.

Beispiel 3:

```
Wunsch = ( "Guten Appetit"| KlammerAusdruck ).
KlammerAusdruck = "(" Ausdruck ")" .
Ausdruck = "Mahlzeit" .
```

Ein Wunsch kann laut dieser Definition entweder aus der Zeichenfolge „Guten Appetit" oder der Zeichenfolge „(Mahlzeit)" bestehen.

Die bisher erwähnten Beschreibungsmöglichkeiten für die Syntax einer Sprache würden schon ausreichen, aber um Schreibarbeit zu ersparen, wurden noch einige Abkürzungsmöglichkeiten geschaffen.

Steht auf der rechten Seite einer Definition ein Ausdruck in eckigen Klammern, so bedeutet dies, daß der Inhalt der Klammern stehen kann oder auch nicht.

Beispiel 4:

```
Satz : "Jetzt geht es " ["endlich "] "zur Sache."
```

Die beiden korrekten Zeichenfolgen wären : „Jetzt geht es zur Sache." und „Jetzt geht es endlich zur Sache."

Eine letzte Schreibweise bedient sich der geschweiften Klammern. Ein Ausdruck in geschweiften Klammern kann beliebig oft oder auch gar nicht stehen.

Beispiel 5:

```
Satz : "Hallo " {"Echo "} "."
```

Korrekte Zeichenfolgen wären : „Hallo .", „Hallo Echo .", „Hallo Echo Echo ." usw..

Mit den obigen Erläuterungen sollten Sie nun den Aufbau der Definitionsdatei verstehen können. Falls Ihnen bisher alles zu theoretisch trocken vorkam, dann sollten Sie sich einfach mal die Definitionsdatei „Edward.DEF" anschauen. Die Struktur, die die Definitionen haben müssen, sollte Ihnen auch durch einfaches Anschauen schnell klarwerden.

3.16.2 Definition eigener Menüs

Die Syntax der Menü-Definitionen ist wie folgt definiert.

```
Menu        = "MENU" MenuTitle { MenuGroup } .
MenuGroup   = "TITLE" QuotedString { MenuItem } .
MenuItem    = ("ITEM" QuotedString CommandList [QuotedString] |
              "EMPTY" ) .
CommandList = ( "(" Command ")" | "(" {CommandList} ")"  ) .
Command     = ( EdwardCommand | CommandList ) .
```

QuotedString ist eine Zeichenfolge zwischen zwei doppelten Anführungszeichen (z.B. „Projekt" für das Projekt-Menü).

EdwardCommand ist ein Kommando des Editors mit Argumenten (falls erforderlich), wie es im Anhang „Die Kommandos des Editors" beschrieben wird.

Zur Bedeutung der einzelnen Definitionen :

Menu:

Die Menüdefinitionen in der Datei „Edward.DEF" müssen mit dem Schlüsselwort **„MENU"** beginnen. Daraufhin folgen die Definitionen der Einträge (MenuGroup), wobei mindestens ein Eintrag definiert werden muß.

MenuGroup:

Eine Menü-Gruppe beginnt mit dem Schlüsselwort **„TITLE"**, dem sich ein QuotedString anschließt. Die Zeichenfolge des QuotedString (s.o.) gibt den Titel der Menü-Gruppe an, der erscheint, wenn Sie die rechte Maustaste drücken.

Auf die Angabe des QuotedSting können beliebig viele Einträge folgen.

MenuItem:

Ein Eintrag in einer Menü-Gruppe (MenuGroup) besteht entweder aus dem Schlüssel-Wort **„EMPTY“**, dieses dient dazu verschieden Einträge in einer Gruppe voneinander durch einen Balken zu trennen, oder dem Schlüsselwort **„ITEM“** und einigen weiteren Angaben. **„ITEM“** muß in jedem Fall ein QuotedString folgen. Dieser gibt den Text des Menü-Eintrags an, der erscheint, wenn Sie mit der Maus die entsprechende Menü-Gruppe aufklappen lassen. Daran schließt sich eine CommandList an. Die CommandList ist eine Folge von Kommandos mit Argumenten, die Edward versteht. Diese werden jedesmal ausgeführt, wenn Sie den entsprechenden Eintrag in einer Menü-Gruppe mit der Maus angewählt haben.
Mit einem weiteren wahlweisen QuotedString können Sie ein Zeichen angeben, welches als Short-Cut dienen soll.

3.16.3 Definition der Tastaturbelegung

```
Keys          = "KEYS" { ModeList }
ModeQualList = "("{ Mode | Qualifier | ""}   { KeyDef }")"
Mode          = ( "MODE1" | "MODE2" | "MODE3" )
Qualifier     = ( "ALT" | "SHIFT" | "CTRL" | "LEFT_COMMAND" )
KeyDef        = ( KeyName | QuotedString | "NP" QuotedString )
                CommandList .

KeyName = (
"F1" | "F2" | "F3" | "F4" | "F5" | "F6" | "F7" | "F8" | "F9" | "F10" |
"ESC" | "BACK" | "DEL" | "HELP" |
"CRSRUP" | "CRSRDOWN" | "CRSRRIGHT" | "CRSRLEFT" |
"TAB" | "RETURN" | "SPACE" | "ENTER" |
"NP_NUML" | "NP_SCRL" | "NP_PRTSC" | "NP_HOME" | "NP_UP" | "NP_PGUP" |
"NP_LEFT" | "NP_RIGHT" | "NP_END" | "NP_DOWN" | "NP_PGDN" | "NP_INS" |
"NP_DEL" )


CommandList  = ( "(" Command ")" | "(" {CommandList} ")"  )
Command      = ( EdwardCommand | CommandList )
```

QuotedString ist eine Zeichenfolge zwischen zwei doppelten Anführungszeichen

(z.B. “Projekt“ für das Projekt-Menü).

EdwardCommand ist ein Kommando des Editors mit Argumenten (falls erforderlich), wie es im Anhang) „Die Kommandos des Editors“ beschrieben wird.

Wie ein Tastendruck interpretiert werden soll, hängt von verschiedenen Umständen ab. Wichtig ist zunächst einmal, ob die Taste in Kombination mit <ALT>, <CTRL>, <SHIFT> oder <LEFT_COMMAND> (der linken <AMIGA>-Taste) betätigt wurde. <ALT>, <CTRL>, <SHIFT> und <LEFT_COMMAND> werden auch als Qualifier bezeichnet. Für jede beliebige Kombination aus diesen Qualifiern wird der Tastendruck gesondert interpretiert. Das bedeutet,

wenn Sie <ALT>+<F1> drücken, geschieht etwas anderes, als wenn Sie <ALT>+<CTRL>+<F1> drücken. Drücken Sie eine Taste, ohne gleichzeitig eine oder mehre Qualifier-Tasten gedrückt zu haben, so wird auch das gesondert interpretiert. Daraus ergeben sich sechzehn verschiedene Möglichkeiten eine Taste zu belegen (2 * 2 * 2 *2; (mit <ALT> und ohne <ALT>) * (mit <CTRL> und ohne <CTRL>) * (mit <SHIFT> und ohne <SHIFT>) * (mit <LEFT_AMIGA> und ohne <LEFT_AMIGA>)).

Das war schon ganz schön kompliziert, aber es wird noch viel schlimmer.

Mit den Befehlen *„SetMode1"*, *„SetMode2"* und *„SetMode3"* können Sie einen speziellen Modus setzen. Ist dieser Modus gesetzt, so wird eine daraufhin gedrückte Taste in Kombination mit allen Qualifiern wieder speziell interpretiert. Danach wird der Modus wieder zurückgesetzt.

Wurde eine Taste nicht mit einer CommandList belegt, so wird beim Druck der Taste das ihr entsprechende ASCII-Zeichen in den Text eingefügt. Das bedeutet für Sie eigentlich nur, daß Sie ein „a" bekommen, wenn Sie die Taste mit dem „a" drücken. Es sei denn, aus irgendwelchen Gründen hätten Sie diese Taste mit einer CommandList belegt.

Was das alles zu bedeuten hat, muß Ihnen jetzt noch nicht sonnenklar aufgegangen sein, zumindest können Sie jetzt einen Zusammenhang mit der oben angegebenen Syntax erkennen, den ich Ihnen aber gerne noch etwas näher erläutere.

Keys:

Jede Definition der Tastaturbelegung muß mit dem Schlüsselwort **„KEYS"** beginnen. Diesem schließt sich eine beliebige Anzahl von ModeLists an.

ModeQualList:

Bei diesem Wort sollte es jetzt bei Ihnen klingeln. In einer ModeQualList geben Sie an, wie eine Taste in Verbindung mit den gesetzten Modes und Qualifiern interpretiert werden soll. Nach der öffnenden runden Klammer "(" mit der eine ModeQualList beginnt, folgt eine beliebige Kombination aus Mode- und Qualifier-Angaben (z.B.: "(MODE2 ALT SHIFT ..." oder gar nichts.
Es schliessen sich die KeyDefs an und zum Schluß kommt die obligatorische schliessende runde Klammer ")".

KeyDef:

Eine KeyDef beginnt mit der Angabe der zu belegenden Taste. Ist dies eine Sondertaste, so geben Sie deren speziellen Namen an (alle Namen sind oben aufgeführt worden). Ansonsten müssen Sie in einem QuotedString das Zeichen angeben, das erscheint, wenn Sie die Taste ohne irgendeinen Qualifier drücken (also ohne <SHIFT>, <ALT>, <CTRL>). Befindet sich die Taste auf dem numerischen Ziffernblock, so geben Sie vor dem QuotedString das Schlüsselwort NP (NP für NumericPad = Zahlenblock) an.
Der Angabe der zu belegenden Taste folgt eine CommandList. Diese gibt die Kommandos an, die Edward nach dem Drücken der entsprechenden Taste ausführen soll.

Damit Sie nun nicht glauben, nur Nobelpreisträger könnten die Tastaturbelegung Edwards ändern, werde ich Ihnen nun am Ende des Abschnittes ein paar Beispiele vorführen.

Beispiel 1 : Zusammenhang zwischen Qualifiern und Taste

```
KEYS
(
     DELETE       (delete)
)
(ALT CTRL
     DELETE (insertchars "Auf 'nem PC wär das das Ende" )
)
```

Mit der obigen Definition wird beim Drücken von DELETE (Taste <DEL>) in Verbindung mit <ALT> und <CTRL> der Text **„Auf 'nem PC wär das das Ende"** ab der aktuellen Cursor-Position eingefügt. Dieses Beispiel befindet sich schon in der Datei „Edward.DEF". Probieren Sie es ruhig mal aus. Drücken Sie dagegen nur <DEL>, so wird das Zeichen rechts vom Cursor gelöscht.

Beispiel 2 : Zusammenhang zwischen Modi und Taste

```
KEYS
(CTRL
   "x"     (SetMode1)
)
(MODE1 CTRL
   "c"  (insertchars   "EMACS Quit" )
   "s"  (insertchars   "EMACS Save" )
)
```

Obige Defnition bewirkt folgendes :

Wenn Sie <CTRL>+<x> drücken wird Modus 1 eingestellt. Die nächste Taste wird also gemäß den Angaben für Modus 1 interpretiert. Danach wird automatisch wieder der Normal-Modus eingestellt. Wenn Sie also <CTRL>+<x> und gleich anschließend <CTRL>+<C> drücken, so wird er Text „EMACS Quit" rechts vom Cursor eingefügt.

Auch dieses Beispiel ist schon in der Datei „Edward.DEF" enthalten. Sie können es daher gleich ausprobieren.

Sollten immer noch nicht alle Klarheiten beseitigt sein, dann laden Sie die Datei „Edward.DEF" einfach ein und schauen sich die Definitionen an, verändern Sie auch ruhig ´mal etwas, aber vergessen Sie nicht, sich eine Sicherheitskopie von „Edward.DEF„ anzulegen.

3.16.4 Definition des HotKeys

```
HotKey     =   "HOTKEY" Qualifier HotKeyDef
Qualifier  =   ( Qualifier | "ALT" | "SHIFT" | "CTRL" |
               "LEFT_COMMAND" | "" ) .
HotKeyDef  =   ( KeyName | QuotedString | "NP" QuotedString ) .
```

```
KeyNam    =  (
"F1" | "F2" | "F3" | "F4" | "F5" | "F6" | "F7" | "F8" | "F9" | "F10" |
"ESC" | "BACK" | "DEL" | "HELP" |
"CRSRUP" | "CRSRDOWN" | "CRSRRIGHT" | "CRSRLEFT" |
"TAB" | "RETURN" | "SPACE" | "ENTER" |
"NP_NUML" | "NP_SCRL" | "NP_PRCSC" | "NP_HOME" | "NP_UP" | "NP_PGUP" |
"NP_LEFT" | "NP_RIGHT" | "NP_END" | "NP_DOWN" | "NP_PGDN" | "NP_INS"
"NP_DEL" ).
```

Wie Sie im Kapitel 3.13 „Iconify & HotKey" nachlesen konnten, kann Edward nach Ausführen der Iconify-Funktion wieder durch einen HotKey aktiviert werden. Welche Taste als HotKey dient, wird ebenfalls in der Datei „Edward.DEF" angegeben. Die Syntax hierfür ist sehr einfach. Die Definition des HotKeys beginnt mit dem Schlüsselwort **„HOTKEY"**. Es folgt eine Liste von Qualifiern wie z.B. <ALT> oder <LEFT_AMIGA> und schließlich wird die Taste angegeben, die hinfort die Rolle des HotKeys spielen soll.

Beispiel:

```
HOTKEY
   CTRL ALT DELETE
```

Das bedeutet, wenn Sie Edward in den Iconify-Zustand geschickt haben, so können Sie Ihn durch Drücken von <CTRL>, <ALT> und der <DEL>-Taste wieder aktivieren. Auf einem PC würde diese Tastenkombination übrigens einen Soft-Reset ausführen.

3.16.5 Einladen einer neuen Definition

Normalerweise lädt Edward die Definition der Menüs, der Tastatur, etc. gleich beim Start aus der Datei „Edward.DEF". Sie können aber auch später die einzelnen Definitionen ändern, indem Sie eine andere Definitionsdatei einladen. Dies ist durch Wahl von **„Def-Datei laden"** aus dem „Extras"-Menü möglich. Es erscheint dann ein Dateiauswahlfenster, mit dem Sie eine andere Definitionsdatei angeben können.

3.17 ARexx - Kommunikation mit Edward

Diese Kapitel kann leider (für Sie) und zum Glück (für mich, den Autor dieses Handbuches) keine Einführung in die Programmiersprache Rexx sein. Dies würde den Rahmen dieses Handbuches bei weitem sprengen.

Außerdem gibt es inzwischen genügend Sekundärliteratur zu diesem Thema. Was soll dann dieses Kapitel, werden Sie sich fragen?

Ganz einfach: Ihnen die Möglichkeiten im Zusammenspiel zwischen Edward und ARexx aufzeigen.

ARexx wurde ja gerade dazu geschaffen, die unterschiedlichsten Programme von außen zu steuern und in ihren Ablauf einzugreifen.

Edward hat auch auf diesem Gebiet eine ganze Menge zu bieten. Das Meiste verbirgt sich jedoch unter der Oberfläche.

Die einzige Funktion, die Sie direkt über Menüs ansprechen können, ist der Menüpunkt „ARexx-Befehl" aus dem „Makro"-Menü. Wenn Sie diese Funktion wählen, so erscheint ein Dateiauswahlfenster, mit welchem Sie ein ARexx-Programm auswählen können. Dieses wird daraufhin ausgeführt.

Alles Weitere, was sich aus dem Zusammenspiel zwischen ARexx und Edward ergibt, wird erst über die speziellen Kommandos Edwards möglich, mit denen intern alle Funktionen, die Sie z.B. über die Menüs aktivieren, ausgeführt werden. Alle diese Kommandos können auch über ARexx angesprochen werden. Dazu ist es notwendig, in Ihrem ARexx-Programm die folgende Zeile einzufügen :

```
ADDRESS "EDWARD"
```

Alle folgenden Befehle, die ARexx selbst nicht interpretieren kann, werden dann an Edward geschickt. Welche Befehle dies sein können, erfahren Sie im Anhang) „Die Kommandos des Editors". Die Befehle in diesem Anhang sind thematisch geordnet. Im Abschnitt XIV. des Anhangs A) finden Sie zudem spezielle Kommandos, die nur im Zusammenspiel mit ARexx sinnvoll sind.

3.18 Zurücknehmen von Text-Änderungen

Mit den Funktionen **„Undo"** und **„Redo"** erhalten Sie durch Edward die Retter in der Not, die Sie wahrscheinlich schon oft vermißt hatten. Wie oft hat man doch schon an einem Text etwas geändert, schnell mal was umkopiert oder gelöscht und dann festgestellt, daß das eigentlich nicht gerade die beste Idee war. Was dann? Natürlich könnte man den alten Text wieder einladen und von Neuem beginnen. Wie lästig.

Zum Glück gibt es die Undo-Funktion in Edward. Damit können Sie eine von Ihnen vorgegebene Anzahl aller Operationen, die einen Text verändert haben, wieder rückgängig machen.

Mit „Redo" ist es Ihnen sogar möglich, die Rücknahme der Änderungen wieder zurückzunehmen (ganz schön komplex).

Natürlich macht Edward das Ganze nicht umsonst.

Um sich alle Änderungen merken zu können, braucht er Speicher. Wieviel, daß können Sie mit den „Globale Einstellungen" ebenso wie die Anzahl der zu merkenden Operationen einstellen.

☞ Siehe hierzu insbesondere Kapitel 3.15 Globale Einstellungen.

Um einen kleine Eindruck der „UnDo"-Funktion zu erhalten, probieren Sie einfach das folgenden Beispiel aus.

- Geben Sie folgende Zeile ein: „Das Nichts als Negation des Seins mit dem Potential der Realisation der Existenz."
- Wählen Sie nun **„UnDo"** aus dem „Extras"-Menü.

Daraufhin verschwindet der letzte Buchstabe, den Sie getippt hatten.

Sie können dies wiederholen, bis irgendwann die Meldung „Everything is undone!" in der Titelzeile erscheint.

- Wählen Sie nun **„ReDo"** aus dem „Extras"-Menü. Daraufhin erscheint der Buchstabe wieder, der durch die vorige Undo-Operation gelöscht wurde.
- Auch **„ReDo"** können Sie wiederholen, bis wieder die gesamte Zeile dasteht.

Vielleicht laden Sie jetzt mal einen größeren Text ein, schneiden Blöcke aus und kopieren Sie um, löschen einige Zeilen und tippen neue Zeichen ein.

Wenden Sie dann die Undo-Funktion an.

Hinweis

Auch die Undo-Funktion stößt einmal an ihre Grenzen. Wenn sehr viel Speicher zum Merken von Textveränderungen benötigt wird, dann kann es vorkommen, daß Edward Speicher, der zum Merken älterer Veränderungen benötigt wurde, freigeben muß. Diese Änderungen sind dann nicht mehr rückgängig zu machen.

Weiterhin können nur so viele Änderungen gemerkt werden, wie Sie in den „Globalen Einsteller" als Undo-Tiefe angegeben haben. Kommen mehr Änderungen zusammen, so werden die ältesten Änderungen vergessen.

4. Syntax

4.1 Allgemeines

4.1.1 Syntaktische Notation

In diesem Handbuch wird als Notation für formale Syntaxdefinitionen EBNF, Niklaus Wirths „Extended Backus Naur Form", verwendet:

`"Abc"`	Der Text zwischen den Anführungszeichen ist so zu übernehmen.
`Abc`	Entweder ist hier eine andere Syntax dieses Namens einzusetzen, oder im nachfolgenden Text steht eine Erläuterung zu diesem Punkt.
`{...}`	Der Inhalt der Klammer kann beliebig oft, inklusive Null-mal, gesetzt werden.
`[ ... ]`	Der Inhalt der Klammern ist optional, d. h. er kann entweder genau einmal gesetzt oder ersatzlos weggelassen werden.
`(...)`	Runde Klammern dienen zur Gliederung und Strukturiereung des Ausdrucks.
`A | B`	Alternative: Entweder A oder B

Ein Beispiel:

```
"(" Zahl { "," Zahl } ")"
```

Dies ist eine geklammerte Liste von wie auch immer gearteten Zahlen: Am Anfang muß immer eine runde Klammer „(„ stehen, gefolgt von einer ersten Zahl. Danach kann man beliebig oft jeweils ein Komma „," und eine Zahl schreiben, bevor man am Ende wieder eine Klammer „)" setzt.

Folgende Zeilen würden also der Syntax genügen:

```
(17,4)
(0)
(1, 2 , 3 , 1,-5)
```

Wie Sie sehen, dürfen Sie auch Leerzeichen einschieben, obwohl diese nicht in der formalen Syntax enthalten sind. Es gibt oft auch den umgekehrten Fall, wie bei der folgenden Syntax (die übrigens kein sinnvoller Teil von Pascal ist):

```
"BEGIN" (Bezeichner | Zahl) "END"
```

Hier ist zwischen „BEGIN" und „END" wahlweise ein Bezeichner oder eine Zahl zu setzen. Hier MUSS man sogar zusätzliche Leerzeichen einfügen:

```
BEGIN42END
```

entspräche nicht der Syntax, da dies für den Compiler ein einziger langer Bezeichner wäre.

Wie ist dieses Problem formal zu lösen? Schließlich wollen wir nicht ständig in Syntaxnotationen Leerzeichen und sonstige Trennungen einfügen.

Der Compiler analysiert den Quelltext auf zwei Ebenen: der lexikalischen und der syntaktischen. Auf der lexikalischen Ebene wird der Text in eine unstrukturierte Folge von Grundsymbolen unterteilt.

Solche Grundsymbole sind im wesentlichen Bezeichner, Wortsymbole, Integerzahlen, Gleitpunktkonstanten, Stringkonstanten und spezielle Symbole wie z. B. das Semikolon oder der Doppelpunkt. So würde der Compiler bei der lexikalischen Analyse „BEGIN42END" als ein einziges Grundsymbol, nämlich einen Bezeichner, erkennen. Um daraus mehrere zu machen, muß man sie irgendwie trennen. In Pascal dürfen zwischen Grundsymbolen beliebig viele Leerzeichen, Zeilenenden, Kommentare usw. stehen, ohne daß der Sinn des Textes dadurch irgendwie beeinflußt wird.

```
BEGIN 42END
```

entspräche unserer Beispiel-Syntax: „BEGIN" ist als erstes Symbol klar zu erkennen. Es folgt eine Zahl, die zwangsläufig da endet, wo keine Ziffer mehr folgt. Somit ist auch das Symbol „END" klar als solches zu identifizieren. Jetzt erst wird die syntaktische Analyse durchgeführt, die die Grundsymbolfolge als korrekt erkennt.

Wenn nicht ausdrücklich anders angegeben, beschreiben EBNF-Ausdrücke in diesem Handbuch Regeln für die syntaktische Analyse, machen also Aussagen über die Abfolge von PASCAL-Grundsymbolen. Dadurch ist also z. B. bei unserer Beispielsyntax

```
"BEGIN" (Bezeichner | Zahl) "END"
```

inhärent klar, daß das „BEGIN" vom nachfolgenden Grundsymbol - hier wahlweise eine Zahl oder ein Bezeichner - getrennt sein muß, und daß andererseits zwischen den Symbolen beliebig viele Leerzeichen, Zeilenenden, Kommentare o. ä. gesetzt werden dürfen. Des weiteren werde ich in Zukunft nicht mehr darauf hinweisen, daß in PASCAL die Groß-Kleinschreibung keine Rolle spielt.

4.1.2 Symbole

Wie oben bereits angedeutet, gibt es in PASCAL Wort- und Spezialsymbole. Zunächst eine Liste der Wortsymbole:

AND	ARRAY	BEGIN	CASE
CONST	DIV	DO	DOWNTO
ELSE	END	FILE	FOR
FUNCTION	GOTO	IF	IN
LABEL	MOD	NIL	NOT
OF	OR	PROCEDURE	PROGRAM
PACKED	RECORD	REPEAT	SET
SHL	SHR	THEN	TO
TYPE	UNTIL	VAR	WHILE
WITH	XOR		

MaxonPASCAL wurde gegenüber dem Standard stark erweitert, was es erforderlich machte, viele Wortsymbole zu ergänzen. Nun können Wortsymbole nicht überdefiniert werden, wodurch es dadurch zu Kompatiblitätsproblemen kommen könnte. Deshalb wurden nur drei „echte" Wortsymbole hinzugenommen, nämlich XOR, SHL und SHR. Alle anderen wurden als sog. Directives realisiert. Sie unterscheiden sich von Wortsymbolen dadurch, daß sie „überdefiniert" werden können. Hier ist eine Liste dieser Directives. Der Jensen-Wirth-Standard kennt übrigens nur eine einzige Directive, nämlich FORWARD.

ABSOLUTE	EXPORT	EXTERNAL	FORWARD
FROM	IMPLEMENTATION	IMPORT	INTERFACE
LIBRARY	MODULE	OTHERWISE	STATIC
STRING	UNIT	USES	

Außerdem gibt es noch Spezialsymbole. Sie bestehen stets aus einem oder zwei Zeichen, die keine Buchstaben sind. Die folgende Liste, die übrigens genau dem Standard entspricht, enthält alle Spezialsymbole:

( ) * + , -. .. / : := ;

< <= <> = > >=

[ ] ^

Ersatzschreibweisen:

(. für [

.) für]

@ als Ersatz für ^ (Jensen-Wirth) ist nicht möglich.

4.1.3 Das Semikolon - das ungeliebte Trennzeichen

Viele genervte Programmierer wird es freuen, daß MaxonPASCAL so gut wie keine Semikola „;" erwartet. Prinzipiell gilt, daß man sie nur da setzen muß, wo sie als Trennzeichen nötig sind. Man sollte sie aber trotzdem setzen (zumal es der „echte" PASCAL-Freak schon fast im Unterbewußtsein tut): Zum einen kann der Compiler dann die Fehler leichter analysieren und zum anderen bleiben die Programme dadurch kompatibel und portabel. Zudem kann und will ich nicht garantieren, daß der Compiler ohne Semikola 100%-ig richtig arbeitet (soll heißen, daß er eventuell hier oder da Fehler melden könnte(!), die keine sind - die Codeerzeugung dürfte dadurch unberührt bleiben).

Mit der Option **„Semikolon melden"** aus dem Compiler-Requester können Sie sich die Stellen, an denen ein „;" fehlt, anzeigen lassen. Die Übersetzung wird dabei nicht abgebrochen.

4.1.4 Kommentare

Kommentare können wahlweise mit "{" oder "(*" beginnen und mit "}" oder "*)" enden. Geschachtelte Komentare sind möglich.

4.1.5 Bezeichner

Bezeichner („Identifier") dienen in PASCAL dazu, Variablen, Prozeduren und vielen anderen Dingen mehr Namen zu geben. Erlaubte Zeichen in Bezeichnern sind: - Buchstaben - Ziffern - der Unterstrich „_" Bei MaxonPASCAL zählen die Umlaute „ä", „ö", „ü" zu den Buchstaben, nicht jedoch das „ß". Das erste Zeichen eines Bezeichners kann ein Buchstabe oder das Unterstreichzeichen sein.

Formale Syntax:

```
Bezeichner   =   Buchstabe { Buchstabe | Ziffer }
Buchstabe    =   'a' | 'b' |  ...  | 'z' | 'ä' | 'ö' | 'ü' | '_'
Ziffer       =   '0' | '1' | '2' |  ...  | '9'
```

Beachten Sie bitte, daß es sich hierbei um eine lexikalische Definition handelt und daß auch hier nicht zwischen Groß- und Kleinbuchstaben unterschieden wird. Was in der Syntax nicht zum Ausdruck kommt: Wortsymbole wie z. B. „BEGIN" sind keine Bezeichner.

4.2 Programmkopf

Die Zeile „Program ...“ kann entfallen. Wenn sie aber gesetzt wird, ist die Syntax folgende:

```
"PROGRAM" Bezeichner [ "(" Bezeichner { "," Bezeichner } ")" ][ ";"s ]
```

Im Klartext: Nach dem Schlüsselwort „Program“ muß stets ein Bezeichner folgen. Optional kann man dann eine in runden Klammern eingeschlossene Parameterliste setzen, bestehend aus durch Komma getrennten Bezeichnern.

Achtung

Der Inhalt dieser Liste wird völlig ignoriert und hat keinerlei Bedeutung! Wie fast immer, ist auch hier das „;“ am Schluß optional.

4.3 Blöcke

Ein Block bildet in PASCAL den Rumpf eines Programms bzw. einer Procedure oder Function und besteht aus:

- **Label-Deklarationsteil**
- **Konstanten-Definitionsteil**
- **Typdefinitionsteil**
- **Variablendeklarationsteil**
- **Procedure- und Function-Deklarationen**
- **Bibliotheksdefinitionen**
- **Anweisungsteil**

Der letzte Teil ist der einzige, der immer vorhanden sein muß, alle anderen sind optional.

Bei MaxonPASCAL ist die Reihenfolge der Deklarations- und Definitionsteile beliebig; jeder Teil kann auch beliebig oft vorkommen. Wichtig ist nur, daß am Ende der mit BEGIN ... END umschlossene Anweisungsteil kommt.

Folgendes wäre ein korrektes MaxonPASCAL-Programm:

```
{ "Program"-Kopf weggelassen }

Const diff=7;

Procedure p(Var i:integer);
  Begin
    writeln(i); i:=i-diff
  End;

Type ttyp=integer;

Var v: ttyp;
```

```
Const con=0815;

Begin
   v:=con; p(v); p(v)
End.
```

Der Punkt am Programmende ist übrigens auch nicht unbedingt nötig.

Formal sieht das Ganze dann so aus:

```
Deklarationsteil = { Labeldeklaration  | Konstantendefinition |
                     Typdefinition  | Variablendeklaration |
                     Prozedurdeklaration  | Librarydefinition

Block            = Deklarationsteil  Anweisungsteil

Programm         = [ Programmkopf ] Uses-Deklaration Block
                   [ "." ]
```

Zu den meisten hier auftretenden Begriffen wie „Variablendeklaration", „Librarydefinition" oder „Anweisungsteil" finden Sie im weiteren Verlauf dieses Handbuchs ausführliche Erläuterungen. Was eine „Uses-Deklaration" ist, erfahren Sie im Kapitel über Units.

4.4 Label

Jedes Label, das benutzt wird, muß zuvor deklariert werden. Die Syntax eines Label-Deklarationsteils ist:

```
Labeldeklaration = "LABEL" Marke { "," Marke }  [ ";" ]

Marke            = Bezeichner | Ziffernfolge
```

Ein Label („Marke") ist entweder eine Integer-Konstante oder ein Bezeichner. Das Label muß auf der Ebene deklariert werden, auf der es gesetzt wird: Man darf z. B. nicht ein Label im Hauptprogramm deklarieren und dann im Anweisungsteil einer Prozedur setzen.

4.5 Konstanten und Konstantendefinition

Syntax für Konstantendefinition:

```
"CONST" Name "=" Konstante { { ";" } Name "=" Konstante } [ ";" ]
```

„Name" ist ein Bezeichner, „Konstante" normalerweise ein Literal. Näheres entnehmen Sie bitte den Kapiteln über die jeweiligen Datentypen.

Eine Konstante kann in MaxonPASCAL aber auch ein Ausdruck sein, in dem nur konstante Operanden und bestimmte Operatoren auftreten. Im Ganzzahl-Bereich können alle einstelligen

(NOT, -) und zweistelligen (+, *, mod, shr, ...) Operatoren in Konstanten benutzt werden, außerdem die folgenden Funktionen (natürlich nur mit konstanten Parametern):

```
Abs   Chr   Ord   Pred   SizeOf   Sqr   Succ
```

Succ, Pred und Ord werden auch bei konstanten Parametern der anderen geordneten Datentypen (Char, Boolean, Aufzählungstypen) ausgewertet.

Real- und String-Ausdrücke werden nicht vom Compiler ausgewertet und können deshalb nicht als Konstante benutzt werden. Die einzigen Operatoren, die bei Real-Konstanten erlaubt sind, sind „+" und „-" als Vorzeichen. Also sind

```
Const  n = 17 + 4;
       nQuadrat = Sqr (n);
       R = 17.4;
       minusR = -R;
       s = '17 plus 4';
```

korrekt definierte Konstanten,

```
Const  r = 17 + 0.4;              { Real-Ausdruck! }
       s = ' Guide' + '42';       { String-Ausdruck! }
```

jedoch nicht.

Außer in Konstantendefinitionen treten Konstanten noch an einigen anderen Stellen auf, z. B. in Typdeklarationen (Ausschnittstypen, z. B. als ARRAY-Indextyp) und als Sprungmarke in der CASE-Anweisung.

Auch hier gelten die selben Regeln für Konstanten-Ausdrücke, so daß also Deklarationen wie

```
VAR Feld: Array[1..40+2] of Set of 0 .. 1 shl 6 - 1;
```

erlaubt sind.

4.6 Typdefinition

In PASCAL kann man sich Datentypen selbst definieren. Um nicht immer wieder dieselbe Typbeschreibung schreiben zu müssen, kann man dem Datentypen einen Namen geben. Dies geschieht im Typ-Definitionsteil eines Blocks: Auf der linken Seite eines Gleichheitszeichens steht der gewünschte Name, auf der rechten Seite die Typbeschreibung. Von dieser Stelle an ist der definierte Typ unter diesem Namen im aktuellen Block verfügbar.

Syntax für Typdefinitionsteil:

```
"TYPE" Name "=" Typ { [ ";" ] Name "=" Typ }  [ ";" ]
```

Den zahlreichen Datentypen ist ein Extra-Kapitel dieses Handbuchs gewidmet.

4.7 Variablendeklaration

Jede Variable, die benutzt wird, muß zuvor deklariert werden. Dies geschieht im Variablen-Deklarationsteil eines Blocks. Jede Variable hat einen Namen, der (natürlich) ein Bezeichner ist, und einen Datentypen, z. B. „Integer", „Real" oder auch ein selbstdefinierter Datentyp.

Zur Syntax:

```
Variablendeklarationsteil     = "VAR" { Variablendeklaration
                              [ ";" ] }

Variablendeklaration          =
     Namensliste ":" Typ  [ [ ";" ]("IMPORT"|"EXPORT"|"STATIC") ]  |
     Name ":" Typ "ABSOLUTE" Konstante

Namensliste                   = Name { "," Name }

Name                          = Bezeichner
```

Es gibt also mehrere Arten von Variablen:

- Die normalen Standardpascal-Variablen. Sie werden durch eine mit „:" abgeschlossene Bezeichnerliste und dahinterstehendem Typ deklariert.

  Beispiel: `a, b, abc: ARRAY [26..731] OF Char;`

  Der Compiler wird angewiesen, zur Laufzeit beim Betreten des Blocks, in dem diese Variablen deklariert sind, auf dem Stack für die Variablen Platz einzurichten.

- Pseudo-Variablen, die an feste Speicheradressen gebunden sind. Hier besteht die Deklaration aus einem einzelnen (!) Bezeichner, dem wie üblich ein Doppelpunkt und ein Datentyp folgen. Dahinter stehen dann (ohne trennendes Semikolon!) die Directive „Absolute" und die gewünschte Speicheradresse in Form einer ganzzahligen Konstanten.

  Beispiel: `Hintergrund: integer ABSOLUTE $dff180;`

  Solche Variablen sind vor allem bei der Programmierung der Hardware interessant: Im obigen Beispiel entspricht die Variable „Hintergrund" dem Speicherwort an der Adresse $dff180 und somit dem Register des Grafikchips, das die Hintergrundfarbe des Bildschirms enthält. Wenn Sie dieser Variablen einen Wert zuweisen, ändert sich die Farbe entsprechend - einmal davon abgesehen, daß das Betriebssystem Ihnen dazwischenpfuscht und in schneller Folge den alten Wert wiederherstellt.

  Es gibt zwei vordefinierte ABSOLUTE-Variablen: „Mem" ist definiert als „Mem: Array [0..MaxLongInt] of Byte Absolute 0", repräsentiert also den gesamten Hauptspeicher, und „SysBase" enthält durch ihre Deklaration als „SysBase: Ptr Absolute 4" einen Zeiger auf die ExecBase-Struktur.

Solche maschinennahen Programmierungen dürften die einzigen sinnvollen Anwendungen von ABSOLUTE-Variablen sein, denn ein anständiger Programmierer benutzt niemals feste Adressen - schon gar nicht auf dem AMIGA!

- Statische Variablen, die im Gegensatz zu den normalen Variablen bereits beim Programmstart eingerichtet werden. Sie belegen keinen Stack-Platz, sondern werden an anderer geeigneter Stelle im Speicher abgelegt. Das macht sich bemerkbar, wenn eine statische Variable lokal in einer rekursiven Prozedur oder Funktion deklariert werden. Bei einer Rekursion wird für sie dann keine „neue" Variable eingerichtet.

  Variablen werden als statisch deklariert, indem man hinter ihre Deklarationszeile die Directive „STATIC" setzt.

  Beispiel:
  ```
  VAR y, x: Real; STATIC;
      s: STRING[20]; STATIC;
  ```

  Im Kapitel „Units und Module" ist den statischen Variablen ein Abschnitt gewidmet.

- Modulüberschreitende importierte oder exportierte Variablen. Diese Variablen sind immer automatisch statisch. Näheres über diese Variablenart finden Sie im Kapitel über Module.

4.8 Prozeduren und Funktionen

PASCAL kennt zwei Arten von Unterprogrammen: Procedures und Functions. Sie dienen jeweils dazu, ein Programm in kleinere Einheiten zu gliedern und so überschaubarer zu machen. Außerdem spart man sich Tipparbeit, wenn man einen öfter identisch oder ähnlich auftretenden Programmteil als Prozedur deklariert. Es gibt aber auch viele andere Probleme, z. B. inhärent rekursive, die man ohne Prozeduren und Funktionen kaum lösen könnte.

4.8.1 Allgemeines über Prozeduren

Eine Prozedur wird im Deklarationsteil eines Blocks deklariert. Sie wird mit dem Prozedurkopf eingeleitet, der im einfachsten Fall aus dem Wortsymbol „PROCEDURE" und dem Namen besteht und mit einem Semikolon abgeschlossen werden sollte:

```
PROCEDURE Beispiel;
```

Danach folgt der Prozedurrumpf, ein Block. Eine Prozedur kann lokale Variablen, Konstanten, Datentypen usw. haben. Es ist sogar möglich, Prozeduren und Funktionen beliebig tief ineinander zu verschachteln (genaugenommen ist diese Schachtelungstiefe in MaxonPASCAL auf die Tiefe 16 beschränkt, aber in der Praxis wird man wohl kaum über eine Tiefe von 3 oder 4 hinauskommen).

Eine Prozedur wird aufgerufen, indem man an der aufrufenden Stelle einfach ihren Namen setzt.

Ein Beispiel:

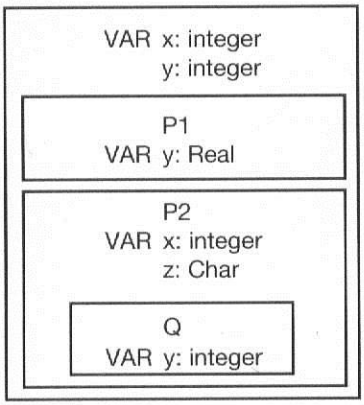
```
PROGRAM ProzedurDemo;

  VAR x, y: integer;

  PROCEDURE P1;
    VAR y: Real;
    BEGIN
      y:= 42;
      writeln ('Prozedur P1', x:10, y:10);
    END;

  PROCEDURE P2;
    VAR x: integer;
    z: Char;

    PROCEDURE Q;  { lokale Prozedur von "P2" }
      VAR y: integer;
      BEGIN
        x:= 1;
        y:= 2;
        z:= '?';
        writeln('Prozedur Q', x:10, y:10, z:10)
      END;        { Ende von "Q" }


    BEGIN  { Anweisungsteil von Prozedur P2 }
      x:= 26;
      y:= 731;
      z:= 'A';
      writeln('Anfang P2', x:10, y:10, z:10);
      Q;
      writeln('P2 nach Q', x:10, y:10, z:10);
      P1;
      writeln('Ende P2  ', x:10, y:10, z:10)
    END;

  BEGIN    { Hauptprogramm }
    x:= 47;
    y:= 11;
    writeln('Programmstart      ', x:10, y:10);
    P1;
    writeln('Mitte Hauptprogramm', x:10, y:10);
    P2;
    writeln('Programmende       ', x:10, y:10)
  END.
```

Dieses Programm gibt folgendes aus:

```
Programmstart          47      11
Prozedur P1            47      42
Mitte Hauptprogramm    47      11
Anfang P2              26      731     A
Prozedur Q             1       2       ?
P2 nach Q              1       731     ?
Prozedur P1            47      42
Ende P2                1       731     ?
Programmende           47      731
```

Wie funktioniert das Ganze? Wir haben ein Hauptprogramm mit zwei Prozeduren, von denen eine wiederrum eine lokale Prozedur enthält. Alle Prozeduren haben lokale Variablen. Man kann sich das so vorstellen:

Beispiel



Zunächst einmal fällt auf, daß Variablen scheinbar doppelt deklariert werden und das auch noch mit verschiedenen Datentypen! Das hängt mit den folgenden Regeln über die Gültigkeitsbereiche von Bezeichnern zusammen.

1. Das Hauptprogramm und alle Prozeduren haben jeweils eine eigene „Schachtel" für ihre Bezeichner. Da Prozeduren wieder selbst als lokale Prozeduren verschachtelt werden können, gibt es eine Hierarchie von Schachteln: Ganz „oben" ist die Schachtel des Hauptprogramms, unter der die Schachteln, der im Hauptprogramm deklarierten Prozeduren stehen, unter welchen jeweils die darin lokal deklarierten Schachteln stehen usw.

2. In jeder Schachtel sind die Bezeichner aus höheren Schachteln bekannt. Sie können aber innerhalb dieser Schachtel „überdefiniert" werden. Eine obere Schachtel merkt grundsätzlich nichts von den Bezeichnern der ihr untergeordneten Schachteln.

3. Ein Bezeichner ist „lokal" zu der Schachtel, in der er deklariert wird und „global" zu allen tiefer liegenden Schachteln. In höheren Schachteln ist er nicht bekannt.

Nun können wir anfangen, das Beispielprogramm Schritt für Schritt durchzugehen. Zunächst wäre da das Hauptprogramm mit den Variablen „x" und „y", jeweils vom Typ „Integer". Von den Variablen, die in den Prozeduren deklariert werden, „weiß" das Hauptprogramm nichts, so daß die Variablen „x" und „y" zunächst ganz normal einen Wert erhalten und ausgegeben werden.

Jetzt wird aber die Prozedur „P1" aufgerufen. Dort ist eine Variable „y" definiert, und es wird die interessante Wertzuweisung „y:= 42" ausgeführt. Die lokale Definition hat Vorrang vor der globalen, also bezieht sich diese Wertzuweisung auf die innerhalb „P1" deklarierte Real-Variable. Diese Variable wird dann auch ausgegeben, ebenso eine Variable „x". Dazu gibt es im Moment nur eine Deklaration: die globale Integer-Variable des Hauptprogramms. Also wird deren Wert (47) ausgegeben.

Nach Ende des Anweisungsteils von „P1" kehren wir in das Hauptprogramm zurück. Die Variablen „x" und „y" haben noch dieselben Werte wie zuvor (47 und 11). In „P1" wurde zwar eine Wertzuweisung an „y" vorgenommen, diese bezieht sich aber auf die dortige lokale Variable, die mit dem gleichnamigen globalen „y" nichts zu tun hat.

Nun wird die Prozedur „P2" ausgeführt. Dort werden als erstes Wertzuweisungen an „x", „y" und „z" durchgeführt. Analog zum ersten Prozeduraufruf können Sie sich klarmachen, daß „x" und „z" lokale Variabeln sind, während „y" sich hier auf den globalen Bezeichner bezieht. Es wird hier also in einer Prozedur der Wert einer globalen Variablen geändert. So etwas nennt man einen „Seiteneffekt" der Prozedur (also eine auf den ersten Blick nicht erkennbare Auswirkung der Prozedur auf den Gesamt-Programmablauf) und sollte möglichst vermieden werden - im nächsten Abschnitt erfahren Sie, wie man so etwas mit Hilfe von „Parametern" sauber machen kann. Jedenfalls wird hier 26, 731 und „A" ausgegeben, bevor „Q" ausgeführt wird. Dort passiert nicht mehr viel Neues: Eine lokale Variable „y" wird eingerichtet, und es gibt einen Seiteneffekt auf die Variablen „x" und „z" der Prozedur „P2". Nach Rückkehr aus „Q" nach „P2" haben diese Variablen also plötzlich einen anderen Wert.

Nun geschieht etwas Interessantes: Die Prozedur „P1" wird aus „P2" heraus aufgerufen. Das ist möglich, denn der Bezeichner „P1" ist ja global, d. h. in der Schachtel des Hauptprogramms, deklariert und somit in der Schachtel von „P2" bekannt. Dort wird wieder die wohlbekannte Zuweisung an das lokale „y" durchgeführt und „x" ausgegeben. Welches „x" ist nun gemeint - das des Hauptprogramms oder das der aufrufenden Prozedur „P2"? In PASCAL gibt es den Begriff des „statischen" und des „dynamischen" Vorgängers einer Prozedur. Statischer Vorgänger von „P1" ist immer das Hauptprogramm, denn darin ist die Prozedur ja deklariert. Dynamischer Vorgänger ist jeweils die aufrufende Schachtel. In diesem Fall, also beim zweiten Aufruf, ist die Prozedur „P2" dynamischer Vorgänger. Die Identifikation von Bezeichnern wird bereits vom Compiler vorgenommen,

muß sich also auf den statischen Vorgänger beziehen, denn der dynamische steht ja erst zur Laufzeit der Prozedur fest. Also bezieht „x" sich hier wieder auf die Variable des Hauptprogramms und nicht die des dynamischen Vorgängers „P2", so daß wieder 47 ausgegeben wird.

Da „P1" keine Seiteneffekte hat, haben sich nach Rückkehr von dort in die Schachtel von „P2" keine Variablenwerte geändert. Nach dem Ende von „P2" und dem Rücksprung ins Hauptprogramm hat es aber einen Seiteneffekt auf die globale Variable „y" gegeben, so daß jetzt ein anderer Wert ausgegeben wird.

In der Praxis wird man natürlich keine so komplizierten Schachtelungen verwenden und nur in Ausnahmefällen globale Bezeichner überdefinieren.

4.8.2 Parameter

Es wurde bereits erwähnt, daß man Seiteneffekte aus Gründen der Lesbarkeit des Listings vermeiden sollte. Um in der Deklaration der Prozedur deutlich zu machen, wovon der Prozedurverlauf abhängt und welche globalen Größen er beeinflußt, gibt es die Parameter.

PASCAL kennt zwei Arten von Parametern: Wertparameter („Call by value") und Variablenparameter („Call by reference"). Andere exotische Programmiersprachen kennen auch „Call by name" (z. B. Simula, Macros in Assembler) oder „Call by value and return" (z. B. Fortran), aber es ist gewiß kein Verlust, daß uns jene Übergabearten in PASCAL erspart bleiben.

Der Witz hat zwar einen Bart, aber hier ist er trotzdem für alle, die ihn noch nicht kennen: PASCAL-Erfinder Niklaus Wirth auf die Frage eines Amerikaners, wie man denn seinen Namen aussprechen solle: „You can call me by name, Wirth, or by value, Worth."

4.8.2.1 Wert-Parameter

Das folgende Programm enthält eine Prozedur, die den größten gemeinsamen Teiler („ggT") zweier Zahlen nach dem Euklidischen Algorithmus berechnet und ausgibt. Damit die Prozedur „weiß", welche beiden Zahlen gemeint sind, werden diese als Parameter übergeben:

```
PROGRAM Euklid;

  PROCEDURE ggT (a, b: integer);
    VAR hilf: integer;
    BEGIN
      WHILE a<>0 DO
      BEGIN
        hilf:= abs (a);
        a:= b MOD a;
        b:= hilf;
      END;
      writeln(b)
    END;

  BEGIN
    write ('Der ggT von 17 und 4 ist ');  ggT (17, 4);
```

```
    write ('Der ggT von 12 und 16 ist ');  ggT (12, 16);
    write ('Der ggT von 42 und 4711 ist ');  ggT (42, 4711);
END.
```

Man kann also in dem Prozedurkopf eine sog. Parameterliste aufnehmen. In unserem Beispiel werden in der Prozedur zwei lokale Variablen „a" und „b" des Typs Integer eingerichtet. Beim Aufruf der Prozedur ist dann anzugeben, mit welchen Werten diese Variablen vorbesetzt werden sollen: Beim ersten Aufruf erhält „a" den Wert 17 und „b" wird 4 zugewiesen usw. Diese Parameter verhalten sich ansonsten wie ganz normale lokale Variablen der Prozedur, z. B. kann man ihnen - wie im Beispiel geschehen - einen neuen Wert zuweisen usw.

Die „aktuellen Parameter" (also die Werte, die man beim Prozeduraufruf angibt) können beliebig komplexe Ausdrücke sein, sie müssen aber zu den „formalen Parametern" (also dem, was in der Prozedurdeklaration in der Parameterliste steht) wertzuweisungskompatibel sein.

Wenn in der formalen Parameterliste mehrere unterschiedliche Parameter stehen sollen, sind diese mit einem Semikolon zu trennen, z. B. so:

```
PROCEDURE MultiPara (Zahl: Real; c1, c2: Char; Flag: Boolean);
```

Beim Aufruf müssen die aktuellen Parameter aber nichtsdestotrotz durch Komma getrennt werden, etwa:

```
MultiPara (x+2, 'A', chr(i+32), b);
```

In der formalen Parameterliste müssen Datentypen grundsätzlich in Form eines Typbezeichners angegeben werden:

```
PROCEDURE Pr(St: STRING[50]);
```

wäre also FALSCH!
„STRING" allein ohne Längenangabe (d. h. mit Default 80) ist streng genommen kein Typbezeichner. Es kann hier aber als solcher verwendet werden.

4.8.2.2 Variablenparameter

Angenommen, wir wollen den ggT nicht ausgeben, sondern weiter mit ihm rechnen. Beispielsweise könnten wir ihn brauchen, um einen Bruch zu kürzen. Es gibt verschiedene Arten, wie eine Prozedur einen Wert zurückgeben kann: Seiteneffekte auf globale Variablen (sollte man besser sein lassen), Funktionen (siehe nächster Abschnitt) und VAR-Parameter.

In unserem ggT-Beispiel könnte das so aussehen:

```
PROCEDURE ggT (a, b: integer; VAR Ergebnis: integer);
  VAR hilf: integer;
  BEGIN
    WHILE a<>0 DO
      BEGIN
        hilf:= abs (a);
        a:= b MOD a;
        b:= hilf;
      END;
    Ergebnis:= b;
  END;
```

Der Variablen-Parameter wird also wie ein Wert-Parameter deklariert, nur daß davor das Schlüsselwort „VAR" gesetzt wird. In der aktuellen Parameterliste ist für den VAR-Parameter eine dazu typkompatible Variable anzugeben, etwa so:

```
VAR n: integer;
...
ggT(42, 4711, n);
writeln('Der ggT von 42 und 4711 ist ', n);
```

Beim Aufruf der Prozedur „ggT" erhält der formale Parameter „Ergebnis" eine „Referenz" auf die aktuelle Parameter-Variable „n". Das bedeutet: eine Variable namens „Ergebnis" gibt es überhaupt nicht, sie ist nur ein anderer Name für den formalen Parameter. Wenn an „Ergebnis" ein Wert zugewiesen wird, erhält in Wirklichkeit der aktuelle Parameter einen neuen Wert. Die beiden Variablen sind für die Dauer der Prozedurabarbeitung identisch.

Ein anderes Beispiel: Eine Prozedur soll einen Bruch, der als Nenner und Zähler übergeben wird, kürzen. Wir verwenden dazu wieder unsere „ggT"-Prozedur.

```
PROGRAM Bruch;

  VAR nenn, zaeh: integer;

  PROCEDURE ggT (a, b: integer; VAR Ergebnis: integer);
    VAR hilf: integer;
    BEGIN
      WHILE a<>0 DO
      BEGIN
        hilf:= abs (a);
        a:= b MOD a;
        b:= hilf;
      END;
      Ergebnis:= b;
    END;

  PROCEDURE Kürzen (VAR zähler, nenner: integer);
    { Die Parameter durch ihren ggT teilen. Wenn nenner=0,
      Fehler melden. Bei zähler=0 den Nenner auf 1 setzen. }
```

```
    VAR g: integer;
    BEGIN
      IF nenner=0 THEN
        error ('Zähler ist Null!')
      ELSE
        IF zähler=0 THEN
          nenner:= 1
        ELSE
          BEGIN
            ggT (zähler, nenner, g);
            nenner:= nenner DIV g;
            zähler:= zähler DIV g
          END
    END;

BEGIN  { Hauptprogramm}
  write('Bruch eingeben: '); readln (zaeh, nenn);
  Kürzen (zaeh, nenn);
  writeln('gekürzt: ', zaeh, ' / ', nenn)
END.
```

Das Programm tut das Gewünschte.

Wir haben es wirklich mit einem „Call by reference" zu tun, nicht mit „Call by name". Beispiel:

```
PROGRAM Kleinlich;
  VAR i: integer;
  a: ARRAY [ 1..2 ] OF integer;

  PROCEDURE Proc (VAR Para: integer);
    BEGIN
      i:= 2;
      Para:= 42
    END;

  BEGIN
    a[1]:= 1; a[2]:= 2;
    i:= 1;
    Proc (a[i]);
    writeln(a[1]: 5, a[2]: 5)
  END.
```

Quizfrage: Was gibt dieses Programm aus?

Antwort: **42**

Beim Aufruf erhält „Para" eine Referenz auf „a[i]" - und das ist zu diesem Zeitpunkt „a[1]". Deshalb wird „42" auch an „a[1]" zugewiesen, obwohl inzwischen „i" einen anderen Wert erhalten hat. Beim sog."Call by name" würde einfach für „Para" jeweils „a[i]" eingesetzt, so daß dann „a[2]" den Wert 42 erhalten würde.

4.8.3 Funktionen

Funktionen sind Prozeduren, die einen Wert zurückgeben. Der Aufruf einer Funktion ist daher kein Befehl, sondern Teil eines Ausdrucks.

Ihre Deklaration ist identisch mit der Prozedur-Deklaration, aber statt „PROCEDURE" ist das Schlüsselwort „FUNCTION" zu verwenden und am Ende des Kopfs ist der Typ des Rückgabewerts anzugeben:

```
FUNCTION ggT (a, b: integer): integer;
  VAR hilf: integer;
  BEGIN
    WHILE a<>0 DO
    BEGIN
      hilf:= abs (a);
      a:= b MOD a;
      b:= hilf;
    END;
    ggT:= b;
  END;
```

Irgendwo im Anweisungsteil der Funktion - oder einer untergeordneten lokalen Prozedur oder Funktion - muß der Rückgabewert definiert werden, indem er dem Funktionsnamen zugewiesen wird.

Ein Aufruf dieser Function könnte so aussehen:

```
writeln('Der ggT von 42 und 4711 ist: ', ggT(42,4711));
```

Oder so:

```
nenner:= nenner DIV ggT(nenner,zähler);
```

Der Rückgabetyp muß in Form eines Typbezeichners angegeben werden. Auch hier ist „STRING" als Pseudo-Bezeichner möglich.

Während Parameter grundsätzlich von jedem beliebigen Typ sein können, sind die Rückgabewerte von Functions auf „skalare" Typen beschränkt. Erlaubt sind:

- **Numerische Datentypen (ganzzahlig und Gleitpunkt)**
- **CHAR, STR, Stringtypen**
- **BOOLEAN**
- **Aufzählungs- und Ausschnittstypen**
- **Pointertypen**

4.8.4 Prozedurparameter und andere Spezialitäten

MaxonPASCAL kennt keine Conformant Array Parameter. Dieses empfohlene Extra des ISO-Standards ermöglicht es, Arrays mit variablem Indexbereich als Parameter zu übergeben. Man kann sich aber mit typfreien Zeigern behelfen (siehe Kapitel 5.6 Pointertypen).

Beispiel:

```
PROGRAM CAP_Ersatz;
  VAR i: integer;
  a: ARRAY[1..10] OF integer;

  PROCEDURE Sort(Arrayzeiger: Ptr; FeldAnzahl: integer);
    VAR ap: ^ARRAY[0..MaxInt] of integer;
    i, j: integer;
    BEGIN
      ap:= Arrayzeiger;
      { Achtung, Array geht von 0 bis Feldanzahl-1 }
      FOR i:= FeldAnzahl-2 DOWNTO 0 DO
        FOR j:= 0 TO i DO
        IF ap^[ j ] > ap^[ j+1 ] THEN
          Exchange(ap^[ j ], ap^[ j+1 ])
    END;

  BEGIN
    FOR i:=1 TO 10 DO a[i]:= random(1000);
    Sort(^a, 10);
    FOR i:=1 TO 10 DO writeln(a[ i ]);
  END.
```

Dieses Programm simuliert einen Conformant Array Parameter, indem ein Zeiger auf das Array (MaxonPASCAL-Spezialität!) und die Anzahl der Elemente übergeben wird. Beachten Sie aber, daß sich der Indexbereich ändert: a[1] entspricht ap^[0].

Man kann auch Prozeduren und Funktionen als Parameter übergeben. Formal sieht das so aus, daß man in die formale Parameterliste einen Prozedur- oder Funktionskopf schreibt. In der aktuellen Parameterliste ist dann nur noch der Name der Prozedur oder Funktion anzugeben.

Beispiel:

```
PROGRAM FuncParam;

  PROCEDURE ab(FUNCTION f(x:Real):Real);
    VAR i:integer;
    BEGIN
      FOR i:=0 TO 10 DO
        writeln(i/5:5:1, f(i/5):12:6);
        writeln;
    END;

  BEGIN
    writeln('Sinus:');   Tab(Sin);
```

```
    writeln('Cosinus:'); Tab(Cos)
END.
```

Dieses Programm druckt Wertetabellen der Funktionen Sin und Cos. Ein weiteres Beispiel zu diesem Thema finden Sie in den Beispielen unter dem Namen „integra.p".

Bei rekursiven Problemen kommt es manchmal vor, daß zwei Prozeduren oder Funktionen sich gegenseitig aufrufen müssen. Die einfachste Lösung wäre, wenn man die Prozeduren ineinander verschachtelt. Falls das aus irgendwelchen Gründen nicht möglich ist, kann man das Problem über „FORWARD"-Referenzen lösen.

Dabei schreibt man von der ersten Prozedur oder Funktion zunächst nur den Kopf, gefolgt von „FORWARD". Nun „weiß" der Compiler, daß es diese Prozedur gibt und wie ihre Parameterliste aussehen soll. Nun kann man die zweite Prozedur und was auch immer schreiben, bevor man irgendwann vor Ende des Deklarartionsteils den Block der ersten Prozedur „nachträgt". Er wird eingeleitet mit dem Kopf dieser Prozedur, aber ohne Parameterliste und (bei Funktionen) Ergebnistyp.

Das folgende Beispiel enthält zwei Funktionen, die auf besonders umständliche Weise feststellen, ob eine Zahl gerade bzw. ungerade ist:

```
PROGRAM Forwarded;

  VAR i: integer;

  FUNCTION Ung(n: integer): Boolean; FORWARD;

  FUNCTION Ger(n: integer): Boolean;
    BEGIN  { "n" ist gerade, wenn n=0 gilt oder n-1 ungerade ist. }
      IF n=0 THEN
        Ger:= true
      ELSE
        Ger:= Ung(n-1)
    END;

  FUNCTION Ung;
    BEGIN  { Eine Zahl ist ungerade, wenn ihr Betrag nicht
             gerade ist. }
      Ung:= NOT Ger(abs(n))
    END;

  BEGIN
    write ('Bitte ganze Zahl eingeben: ');
    readln (i);
    IF ger(i) THEN
      writeln('Gerade.')
    ELSE
      writeln('Ungerade.')
  END.
```

4.8.5 Zusammenfassend: Syntax

Hier ist die vollständige Syntax für Prozedur- und Funktions-Deklarationen:

```
Prozedur/Funktionsdeklaration
                =    ProzFunkKopf (("FORWARD" | "IMPORT" |
                     "EXTERNAL") ";" | ["EXPORT" ";"]  Block)

ProzFunkKopf    =    ("PROCEDURE" Bezeichner [ Formale
                     Parameterliste ] [ ";" ])  | ("FUNCTION"
                     Bezeichner [ FormaleParameterliste ]
                     [ ":"   Typbezeichner ] [ ";"  ])

FormaleParameterliste
                =    "(" FormalerParameter { ";" Formaler
                     Parameter } ")"

FormalerParameter
                =    ([ "VAR" ] Bezeichner { "," Bezeichner } ":"
                TypBezeichner)  | ProzFunkKopf
```

Zu „IMPORT", „EXPORT" und „EXTERNAL" finden Sie im Linker-Kapitel nähere Erläuterungen.

4.9 Anweisungsteil

Der Anweisungsteil eines Blocks besteht aus einer mit BEGIN...END geklammerten Folge von Befehlen. Er ist der einzige Teil, der in einem Block nie fehlen darf.

Syntax:

```
Anweisungsteil =  "BEGIN"  Anweisung { [ ";" ] Anweisung }  "END"
```

Es gibt viele verschiedene Arten von Anweisungen, was auch in unserer EBNF-Syntax zum Ausdruck kommt:

```
Anweisung =  [ Marke ":" ] [ Wertzuweisung | Prozeduraufruf |
             Verbundanweisung | Fallunterscheidung | REPEAT-Schleife |
             WHILE-Schleife | FOR-Schleife | CASE-Anweisung |
             WITH-Anweisung | GOTO-Sprung  ]
```

Vor jeder Anweisung kann also eine Sprungmarke stehen, und eine Anweisung kann grundsätzlich auch leer sein.

4.9.1 Wertzuweisung

Syntax: `Variable ":=" Ausdruck`

Der Ausdruck muß zur Variablen wertzuweisungskompatibel sein. Dabei gelten folgende Regeln:

Variable	Ausdruck
Ganzzahlig	beliebiger Ganzzahltyp, auch andere „Breite"
Real, Double	numerisch
Char	Char-Konstante, -Variable, -Funktion
Boolean	Boolean
Aufzählungstyp	Ausdruck desselben Aufzählungstypen
Pointer	typfreier Zeiger, „NIL" oder Zeiger mit typkompatiblem
Referenztyp	
Str	Char- oder Str-Ausdruck, STRING-Variable, String-Konstante, keine String-Ausdrücke!
SET	beliebiger Mengenausdruck
STRING[n]	Char- oder Stringausdruck
ARRAY	Array-Variable mit gleichem Indextyp und typkompatiblem
Elementtyp	
RECORD	gleicher Recordtyp, oder strukturgleicher Record mit jeweils typkompatiblen Feldern

In der Tabelle taucht des öfteren der Begriff der „Typkompatibilität" auf. Diese Kompatibilität ist strenger definiert als der Begriff der Wertzuweisungskompatibilität: hier müssen skalare Typen zusätzlich die gleiche „Breite" haben. Falls Sie in der Übersicht die Ausschnittstypen vermißt haben: Sie werden mit ihren jeweiligen Grundtypen identifiziert. Ein Datentyp wie „1..10000" wäre also sowohl wertzuweisungs- als auch typkompatibel zu „Integer".

4.9.2 Prozeduraufruf

Es gibt vordefinierte Standardprozeduren (siehe Kapitel 7) und selbstdefinierte Prozeduren, wie im Abschnitt 8 dieses Kapitels dargestellt wurde. Beim Aufruf sind diese beiden Prozedurtypen völlig gleichgestellt:

```
Prozeduraufruf= ProzedurBezeichner [ Parameterliste |
                                     Write-Parameterliste e ]

Parameterliste= "(" Ausdruck { "," Ausdruck } ")"
```

Abgesehen von „Write" und „WriteLn" sehen Parameterlisten also stets gleich aus. Allerdings ist es bei vordefinierten Prozeduren oft so, daß der genaue Inhalt von Parameterlisten, z. B. Anzahl oder Datentyp der Parameter, nicht fest ist.

Alles über vordefinierte Prozeduren finden Sie im Kapitel 7.

4.9.3 Verbundanweisung

Man kann mehrere Anweisungen zu einer einzigen zusammenfassen, indem man sie mit „BEGIN" ... „END" klammert:

```
Verbundanweisung = "BEGIN" Anweisung { [ ";" ] Anweisung } "END"
```

Vor dem „END" muß also kein Semikolon stehen. Man darf es natürlich setzen, was insofern in der Syntax enthalten ist, da man sich als letzte Anweisung eine Leeranweisung denkt.

Verbundanweisungen werden bei den im folgenden beschriebenen strukturierten Anweisungen benötigt, da an einigen Stellen formal keine Anweisungsfolge, sondern eine einzelne Anweisung stehen muß.

4.9.4 Fallunterscheidung

Die wichtigste Art der Fallunterscheidung ist die „IF"-Anweisung. Ihre Syntax ist wie folgt:

```
"IF" Ausdruck "THEN" Anweisung [ "ELSE" Anweisung ]
```

Als Bedingung ist ein beliebiger Ausdruck des Typs „Boolean" erlaubt. Insbesondere kann hier eine einzelne Variable dieses Typs gesetzt werden. Ich weise darauf extra hin, da es ein „beliebter" Stil-Fehler ist,

```
IF b = true THEN ...
```

zu schreiben (wobei „b" eine boolesche Variable ist). Hier reicht ganz einfach:

```
IF b THEN ...
```

Ist der Ausdruck „wahr", wird die Anweisung hinter dem „THEN" ausgeführt (mit Hilfe einer Verbundanweisung kann man hier natürlich auch eine beliebig lange Anweisungsfolge setzen). Ist die Bedingung nicht wahr, so wird - falls vorhanden - die Anweisung des „ELSE"-Teils ausgeführt.

Es gibt eine kleine „Zweideutigkeit" in diesem Zusammenhang:

```
IF   b1 THEN
     IF b2 THEN s1
     ELSE s2;
```

Diese Zeile ist - syntaktisch betrachtet - nicht eindeutig: Das „ELSE" könnte theoretisch zu beiden „IF" gehören. Man hat hier definiert, daß es zum letzten „IF" gehört. Die obige Anweisung entspricht also

```
IF   b1 THEN
     BEGIN
        IF b2   THEN s1
                ELSE s2
     END;
```

Um das „ELSE" ausdrücklich an das erste „IF" zu binden, könnten Sie folgendes schreiben:

```
IF   b1 THEN
     BEGIN
       IF b2 THEN s1
     END
   ELSE
     s2;
```

4.9.5 REPEAT-Schleife

Syntax:

```
"REPEAT" Anweisung { ";" Anweisung } "UNTIL" Ausdruck
```

Der Schleifeninhalt wird ausgeführt, bis die Bedingung (beliebiger Boolean-Ausdruck) hinter dem „UNTIL" wahr ist. Da die Bedingung am Schleifenende geprüft wird, wird die Schleife mindestens einmal durchlaufen.

4.9.6 WHILE-Schleife

Syntax:

```
"WHILE" Ausdruck "DO" Anweisung
```

Die Schleife wird durchlaufen, solange die Bedingung erfüllt ist.

Unterschiede zu „REPEAT":

- Die Bedingung (Boolean-Ausdruck) wird hier am Schleifenanfang geprüft. Dadurch wird die Schleife überhaupt nicht durchlaufen, wenn die Bedingung schon vor dem ersten Durchlauf „false" ist.
- Die Schleife bricht ab, sobald die Bedingung NICHT erfüllt ist (die REPEAT-Schleife wird durchlaufen, bis die Abbruchbedingung erfüllt ist).
- Der Schleifenkörper der WHILE-Schleife besteht formal nur aus einer einzigen Anweisung. In Form einer Verbundanweisung (mit „BEGIN" und „END" eingeklammert) kann man aber auch hier eine beliebig lange Anweisungsfolge benutzen.

4.9.7 FOR-Schleife

Syntax:

```
"FOR" Variable ":=" Startwert ("TO" | "DOWNTO") Endwert "DO" Anweisung
```

Die Variable wird auf den Startwert gesetzt und dann nach jedem Schleifendurchlauf auf den nächsthöheren (bzw. nächsttieferen, falls „DOWNTO" verwendet wird) Wert gesetzt, bis der Endwert überschritten (bzw. unterschritten) wird.

Als Schleifenzählvariable ist jede Variable eines geordneten Typs erlaubt, das sind:

- **Ganzzahltypen (NICHT Real!)**
- **Char**
- **Boolean**
- **Aufzählungstypen**
- **Ausschnittstypen aus diesen genannten Typen.**

Im Gegensatz zu einigen anderen PASCAL-Implementierungen muß die Variable nicht lokal sein.

Start- und Endwert sind beliebig komplizierte Ausdrücke, die zur Zählvariablen zuweisungskompatibel sein müssen.

Bitte denken Sie daran, daß hinter dem „DO" kein Semikolon stehen darf, es sei denn, Sie wollen ausdrücklich eine leere Schleife programmieren.

Der Rumpf der FOR-Schleife wird keinmal ausgeführt, wenn der Startwert größer (bei „TO") bzw. kleiner (bei „DOWNTO") als der Endwert ist.

Innerhalb der Schleife darf der Wert der Schleifenvariablen nicht verändert werden. Dies wird vom Compiler aber nicht überprüft! MaxonPASCAL berechnet vor dem ersten Schleifendurchlauf, wie oft die Schleife auszuführen und die Zählvariable weiterzuzählen ist.

Die Schleife

```
FOR v:= e1 TO e2 DO s;
```

entspricht in etwa:

```
v:= e1;
count:= ord(e2) - ord(v);
WHILE count >= 0 DO
  BEGIN
    s;                  { Schleifenrumpf ausführen }
    v:= succ (v);       { Zählvariable weitersetzen }
    count:= count - 1 { internen Zähler erniedrigen }
  END;
```

4.9.8 CASE-Anweisung

Syntax:

```
"CASE" Ausdruck "OF"
  { Konstantenliste ":" Anweisung }
  [ ("OTHERWISE" | "ELSE") [ Anweisung { ";" Anweisung } ] ]
"END"

Konstantenliste = Konstante [ ".." Konstante ] { "," Konstante [ ".."
Konstante ] }
```

Die CASE-Anweisung ist eine Mehrfach-Fallunterscheidung, bei der aufgrund des Vergleichs eines geordneten Ausdrucks (also Ganzzahlig, Boolean, Char, Aufzählungstyp) mit Konstanten verzweigt wird.

Wie Sie schon aus der Syntax erkennen können, gibt es hier zwei Ergänzungen zum ISO-Standard:

- Es können Bereiche von Konstanten angegeben werden.

  Beispiel:

```
Case i Of
  -MaxInt..0, 20..MaxInt:
        writeln('Eingabe nicht im erlaubten Bereich.');
  1..8:  writeln('Zu klein!');
  12..19:writeln('Zu groß!');
  9, 11: writeln('Fast richtig.');
  10:    writeln('Richtig.')
End;
```

  Dieses CASE-Statement gibt zu jeder Integerzahl „i" einen Kommentar aus.

- Es gibt einen optionalen ELSE-Zweig, der wahlweise mit „ELSE" oder „OTHERWISE" eingeleitet wird. Dieser Teil kann beliebig viele Anweisungen enthalten.

  Beispiel:

```
Case c Of
  'A', 'a': writeln('Ah');
  'B', 'b': writeln('Bee');
  'C', 'c': writeln('Zeh');
  'D', 'd': writeln('Dee');
  '0'..'9':   Begin
                writeln('Bitte keine Zahlen!');
                writeln('(Ich kann nämlich nicht zählen');
              End;
  'e'..'z': writeln('Irgend so'n Kleinbuchstabe.');
  'E'..'Z': writeln('Das war ein GROSSER Buchstabe.');
Else
  writeln;
  writeln('Tut mir leid, diesen Buchstaben kenne ich nicht.');
End;
```

Wie bereits erwähnt, hätte man statt ELSE auch OTHERWISE schreiben können. Ferner sehen Sie, daß vor ELSE (im Gegensatz zu IF-THEN-ELSE) ruhig ein Semikolon stehen darf.

Hat die CASE-Anweisung keinen ELSE-Zweig und der Ausdruck einen Wert, der in den Konstantenlisten nicht auftaucht, so wird ein Laufzeitfehler gemeldet. Es ist nicht erlaubt, einen Wert mehrfach in den Konstantenlisten aufzuführen. MaxonPASCAL meldet dabei aber keinen Fehler, sondern führt, wenn der Ausdruck einen doppelt vorgesehenen Wert hat, nur die erste Anweisung aus!

4.9.9 WITH-Anweisung

Syntax:

```
"WITH" Variable { "," Variable } "DO" Anweisung
```

Die Variable muß von einem RECORD-Typ sein. Innerhalb der WITH- Anweisung kann man dann jedes Feld dieses Records direkt wie eine Variable ansprechen.

Beispiel:

```
TYPE Person = RECORD
  Name: RECORD
        Vor, Nach: STRING [40]
      END;
  Plz: integer;
  Ort: STRING [50]
END;

VAR P1, P2: Person;

...

WITH P1 DO
  BEGIN
    Name.Vor:= 'Dietrich';
    Name.Nach:= 'Gerstenberger';
    Plz:= 4799;
    Ort:= 'Borchen'
  END;
```

Die Anweisung ließe sich, da „Name" innerhalb der WITH-Anweisung ja eine eigenständige Variable ist, so weiter vereinfachen:

```
WITH P1 DO
  WITH Name DO
  BEGIN
    Vor:= 'Helmut';
    Nach:= 'Kohl';
    Plz:= 5300;
    Ort:= 'Bonn'
  END;
```

Dafür gibt es eine Abkürzung: Man faßt einfach die beiden WITH's zusammen:

```
WITH P1, Name DO
  BEGIN ...
  END;
```

Es ist aber nicht zwingend, daß die zweite Variable Bestandteil der ersten ist, es können auch voneinander unabhängige Variablen sein.

Beispiel:

```
WITH P1, P2.Name DO
  BEGIN
    Nach:= Name.Nach;
    P2.Plz:= Plz;
    P2.Ort:= Ort
  END;
```

Was geschieht hier? Durch „WITH P1" werden die Variablen „Name", „Plz" und „Ort" eingerichtet, die sich auf Bestandteile von „P1" beziehen, während „WITH P2.Name", die zu „P2" gehörigen Variablen „Vor" und „Nach" erzeugt. Also erhält in unserem Beispiel „P2" Nachname, Ort und Postleitzahl von „P1".

4.9.10 GOTO-Sprünge

Syntax:

```
"GOTO" Label
```

Ein Label kann (siehe Kapitel 4.4 „Label") entweder eine Ziffernfolge oder ein Bezeichner sein.

Gemäß ISO-Standard kann man aus Strukturen mit „Goto" herausspringen (z. B. aus Schleifen heraus oder aus Unterprogrammen ins Hauptprogramm), aber nicht umgekehrt.

MaxonPASCAL macht sich aber nicht die Mühe, dies alles genau zu prüfen. So ist es z. B. möglich, in eine WHILE- oder REPEAT-Schleife hineinzuspringen.

Lediglich Einsprünge in FOR-Schleifen und WITH-Anweisungen werden abgefangen (und natürlich der Sprung in ein Unterprogramm - schon allein dadurch, daß ein dort gesetztes Label ja lokal deklariert sein muß).

Allgemein anerkannte Konvention über Goto:

Sprünge sollten nur da eingesetzt werden, wo der normale Programmablauf abgebrochen wird (z. B. wegen eines Eingabefehlers) oder wo das Programm ohne GOTO unnötigerweise übermäßig aufgebläht würde. Man sollte aber bei jedem GOTO einen Kommentar setzen, aus dem hervorgeht, warum und wohin hier gesprungen wird und bei jedem Label vermerken, von wo und unter welchen Bedingungen hierher gesprungen wurde.

5. Die Datentypen von MaxonPASCAL

5.1 Numerische Typen

MaxonPASCAL bietet nicht weniger als 7 verschiedene numerische Datentypen:

Integer
Der gewohnte Ganzzahlbereich -(MaxInt+1)..+MaxInt, MaxInt=32767

LongInt bzw. Long
32-Bit-Ganzzahltyp mit Vorzeichen, Bereich -(MaxLongInt+1) bis +MaxLongInt. Die Konstanten „MaxLongInt" und „MaxLong" haben den Wert 2147483647.

ShortInt bzw. Short
Vorzeichenbehafteter 8-Bit-Integertyp, MaxShortInt=MaxShort=127.

Cardinal bzw. Word
Dieser Ganzzahltyp ist vorzeichenlos, umfaßt also nur positive Werte und zwar im Bereich von 0 bis MaxCard=MaxWord=65535.

ShortCard bzw. Byte
Vorzeichenlose 8-Bit-Zahl, Wertebereich von 0 bis MaxShortCard =MaxByte=255.

Real
32-Bit-Realzahlen: 24 Bit Mantisse, 1 Bit Vorzeichen, 7 Bit Exponent. Das reicht für ca. 7-stellige Rechengenauigkeit im Bereich bis etwas über 8*10^18. Nicht gerade viel, aber für die meisten Zwecke genug.

LongReal bzw. Double
Seit Version 2.0 gibt es jetzt auch Fließpunktzahlen doppelter Genauigkeit nach IEEE-Norm. Sie sind 64 Bits breit (1 Bit Vorzeichen, 11 Bit Mantisse, 52 Bit+1 Hidden-Bit Mantisse), belegen im Speicher also 8 Bytes. Die Rechengenauigkeit ist etwa 15-stellig, der Exponentenbereich ist 1E+-308, das sollte genügen.

5.1.1 Ganzzahl-Typen

Fünf der genannten Zahlentypen, nämlich Integer, ShortInt, LongInt, Cardinal und ShortCard, sind sogenannte Ganzzahltypen, d. h. sie umfassen nur ganzzahlige Werte. Alle diese Typen können beliebig durcheinander verwendet werden, ohne daß man Typwandlungen vornehmen muß. Der Haken: MaxonPASCAL muß trotzdem dafür sorgen, daß verschiedene zusammen benutzte Datentypen aneinander angepaßt werden.

Diese Transformationen kosten teilweise nicht unerheblich viel Laufzeit, so daß es in der Regel besser ist, in Ausdrücken nur Daten gleichen Typs zu benutzen.

MaxonPASCAL wählt bei Kombination von unterschiedlichen Datentypen stets einen höheren, der beide umfaßt, z. B. wird die Summe eines Integer- und eines Cardinalwerts im LongInt-Bereich berechnet.

Ausschnittstypen sind stets Integer- oder LongInt-Typen, auch wenn sie theoretisch in einem „kleineren" Typ Platz hätten. Beispiel:

```
Type
  Hundert = 1..100; {belegt 16 Bit, d.h. ist Ausschnitt von Integer}
  NochMehr = 0..50000 { ist ein LongInt-Typ, obwohl auch Cardinal
                        möglich wäre. }
```

Es können auch Hexzahlen (Beispiel: $686b, $00C00276, $0d, $Bad) und Binärzahlen (Beispiel: %110, %101010101, %0011) verwendet werden.

Bei allen Ganzzahltypen gibt es folgende arithmetische Operatoren:

+	Addition
-	Subtraktion oder auch Vorzeichen
*	Multiplikation
DIV	Ganzzahlanteil der Division
MOD	Divisionsrest

Natürlich gilt „Punktrechnung vor Strichrechnung", so daß Multiplikation und Division Vorrang vor Addition und Subtraktion haben.

Gerechnet wird aber nur in den Wertebereichen Integer, Cardinal und LongInt. Wenn zwei ShortInt- oder ShortCard-Werte verknüpft werden, werden sie automatisch nach Integer oder Cardinal gewandelt.

Die beiden Divisionsoperatoren funktionieren nicht ganz korrekt, wenn einer der Operanden negativ ist: In der Mathematik ist es üblich, daß „a MOD b" stets ein Ergebnis zwischen 0 und dem Betrag von b-1 liefert und daß bei „a DIV b" immer das Ergebnis der normalen Division abgerundet wird.

Die Divisionsbefehle des Mikroprozessors MC680x0, der in Ihrem AMIGA steckt, arbeiten aber anders: Hier wird erst mit den Beträgen der Operanden gerechnet, und anschließend wird das Vorzeichen des Ergebnisses negativ gewählt, wenn genau einer der Operanden negativ war. Aus Geschwindigkeitsgründen muß auch MaxonPASCAL diese eigentlich falschen Operationen benutzen.

Die folgende Tabelle soll diesen Sachverhalt verdeutlichen:

		Mathematisch	MC 680x0
42	DIV 10	4	4
(-42)	DIV 10	-5	-4
42	DIV (-10)	-5	-4
(-42)	DIV (-10)	4	4
42	MOD 10	2	2
(-42)	MOD 10	8	-2
42	MOD (-10)	8	-2
(-42)	MOD (-10)	2	2

So tragisch, wie es zunächst scheint, ist das Ganze aber nicht, denn in der Praxis ist der linke Operand selten und der rechte fast nie negativ. Außerdem ist sowohl bei der mathematisch üblichen als auch der von Motorola (dem 680x0-Hersteller) vorgenommenen Definition gewährleistet, daß stets folgende Gleichung gilt:

```
a = b*(a DIV b) + (a MOD b)
```

Bei arithmetischen Operatoren kann es natürlich zu Überläufen kommen, wenn das Ergebnis einer Rechnung nicht im gewählten Rechenbereich liegt. Es kommt nur dann zu einer Fehlermeldung, wenn die entsprechende Compiler-Option mit „{$opt a+}", dem Pull-Down-Menü „Optionen/Compiler" oder dem „Preferences"-Tastaturmenü eingeschaltet wurde, andernfalls ist das Ergebnis undefiniert. Genaueres über diese Option erfahren Sie im Kapitel 8, Abschnitt 1.5: „Compiler-Optionen".

Im Abschnitt „Typkonvertierungen" dieses Kapitels finden Sie Tips, wie Sie in bestimmten Fällen Überläufe vermeiden können.

Neben den erwähnten arithmetischen gibt es noch die logischen Operatoren AND, OR, XOR, NOT, SHL und SHR, die Zahlen bitweise verknüpfen:

AND, OR und XOR funktionieren im Prinzip genau wie die gleichnamigen Boolean-Operatoren, nur daß hier jeweils die Bits der Operanden zu den entsprechenden Bits des Ergebnisses verknüpft werden.

Hier sind einige Beispiele - mit Binärzahlen, da es hier ja auf die Bits ankommt:

```
      %11111100
AND   %10111010
-------------------
=     %10111000
```

Ein Bit wird gesetzt, wenn beide Bits der Operanden gesetzt sind.

```
      %1100
OR    %10001010
-------------------
=     %10001110
```

Ein Bit wird gesetzt, wenn mindestens eins der Operandenbits gesetzt ist.

	%11001100	Ein Bit wird gesetzt, wenn die
XOR	%11111010	beiden Operandenbits verschieden
		sind, also genau eins gesetzt
=	%00110110	ist.

NOT führt auf ganzen Zahlen eine logische Negation aus, d. h. alle Bits des folgenden Operanden werden invertiert.

Beispiele:

```
NOT 0       = -1
NOT %110    = %1111111111111001 = -7
NOT 26730   = -26731
```

Genau wie bei booleschen Ausdrücken hat NOT auch hier höchste Priorität, so daß

`NOT a*b`	äquivalent zu
`(NOT a) * b`	ist.

Außerdem gibt es noch die Operatoren SHL und SHR, „Shift left" und „Shift right". Sie verschieben die Bits des linken Operanden um so viele Positionen nach links bzw. rechts, wie der zweite Operand angibt. Das entspricht einer Multiplikation mit bzw. einer Division durch eine Zweierpotenz.

Beispiele:

```
1 Shl 6          = 64    (= 1 * 2^6)
10 Shr 1         = 5     (= 10 div 2^1)
MaxInt Shr 14    = 1     (= (2^15-1) div (2^14))
```

Beide Operanden müssen ganzzahlig sein. Der rechte Operand darf nicht negativ sein, andernfalls ist das Ergebnis undefiniert (wenn es sich um eine negative Konstante handelt, z. B. „17 shl -2", meldet bereits der Compiler einen Fehler). Es findet kein Test auf Überlauf statt, auch nicht, wenn die Compiler-Option ***„Arithm. Überl."*** aktiv ist.

Die Shift-Operanden zählen zu den Multiplikationsoperanden, sie haben also dieselbe Priorität wie „*", „DIV" oder „MOD".

Beispiel:

`NOT 1 shl 5 + 2` entspricht `((NOT 1) shl 5) + 2`

Hier ist noch einmal eine Übersicht über alle Ganzzahl-Operatoren und ihre Auswertungsreihenfolge. Es werden immer zuerst die Operatoren in den oberen und dann die in den unteren Zeilen der Tabelle zusammengefaßt. Operatoren, die in der selben Zeile stehen, werden von links nach rechts ausgewertet.

	Arithmetisch	Logisch
Invertierung:		NOT
Multiplikativ:	*, DIV, MOD	AND, SHL, SHR
Vorzeichen:	+, -	
Additiv:	+, -	OR, XOR
Vergleiche:	<, >, <=, >=, =, <>	

Bei der Ausgabe mit „*Write*" werden Integer-Zahlen linksbündig ausgegeben (Abweichung vom Standard!).

```
Write(17, 4, 17+4);
```

erzeugt also die Ausgabe

```
17421
```

Sie erhalten eine rechtsbündige Ausgabe, indem Sie hinter der Zahl die gewünschte Feldbreite als Ganzzahl-Ausdruck angeben:

```
Write(17:5, 4:5, 17+4:10)
```

gibt aus:

```
   17    4         21
```

Bei der Eingabe mittels „*Read*" werden nur dezimale Zahlen akzeptiert. Im Falle einer unerlaubten Eingabe wird aber kein Laufzeitfehler gemeldet.

5.1.2 Gleitpunktzahlen

MaxonPASCAL benutzt die normalen Gleitpunkt-Bibliotheken, die das Betriebssystem zur Verfügung stellt. Insbesondere stehen zwei verschiedene Zahlenformate zur Verfügung: Die einfach genauen „Real"-Zahlen und ein doppelt genaues Format, das „LongReal" oder „Double" heißt.

Eine Real-Zahl ist 32 Bit „breit" und belegt also 4 Bytes im Speicher, während Double 64 Bit=8 Byte breit ist. Rechnungen mit doppelter Genauigkeit sind erheblich langsamer als normale Real-Rechnungen, allerdings unterstützt die Mathebibliothek doppelter Genauigkeit eine evtl. vorhandene FPU (Fließkomma-Coprozessor).

Es ist erlaubt, in Ausdrücken beide Zahlenformate zu kombinieren oder auch Gleitpunktzahlen mit Ganzzahlausdrücken zu verknüpfen. Auch hier gilt, daß Typumwandlungen nicht unerheblich Rechenzeit benötigen.

Auf beiden Zahlentypen gibt es folgende Operatoren:

*	Multiplikation
/	Division
+	Addition
-	Subtraktion oder Vorzeichen

Natürlich gilt auch hier „Punkt- vor Strichrechnung". Ferner gibt es die üblichen Vergleichsoperationen, die einen Boolean-Wert liefern:

=, <>, <, >=, >, <=

Ein Problem der meisten Programmiersprachen ist die „Überladung" von Operatoren: „*", „+" und „-" sowie die Benutzung der Vergleichsoperatoren sowohl für Ganzzahl- als auch für Realoperationen. Dabei gelten folgende Regeln:

- Wenn einer der beiden Operanden von einen Fließkommatyp ist, wird auch im Fließkommabereich gerechnet.
- Bei einer Verknüpfung eines Real- und eines Double-Operanden wird das Ergebnis im Double-Bereich berechnet.

Der Divisionsoperator „/" liefert stets eine Fließkommazahl als Ergebnis, bei „DIV" und „MOD" sind nur ganze Zahlen als Operanden erlaubt.

Die beiden Fließkommatypen sind auch wertzuweisungskompatibel. Wenn man aber einen Real-Wert an eine Double-Variable zuweist, kann man Überaschungen erleben:

```
Var r: Real;
d: Double;
Begin
   r:= 0.2;
   d:= r;
   writeln(r, d)
End.
```

Ergebnis: `0.2  0.1999999880079071`

Die Double-Variable hat also scheinbar einen ungenauen Wert. Wie ist das möglich, da Double doch, Gerüchten zufolge, genauer ist als Real? Nun, eine Zahl wie 0.2 ist binär nicht mit endlich vielen Stellen darstellbar (im Dezimalsystem kann man 1/3 ja auch nicht als endlichen Dezimalbruch darstellen). Wenn diese Zahl nun an die Real-Variable zugewiesen werden soll, erhält die Variable in Wirklichkeit den besten darstellbaren Näherungswert für 0.2 - und das ist in dezimaler Darstellung etwa 0.1999999880079071, Sie sehen hier die etwa 7stellige Genauigkeit. Wenn diese Realzahl nun ausgegeben wird, berücksichtigt die Ausgaberoutine des MaxonPASCAL-Laufzeitsystems die geringe Rechengenauigkeit und rundet auf 6 oder 7 Dezimalstellen, je nach gewünschtem Ausgabeformat. Also wird 0.2 ausgegeben, und der Anwender wird mit diesem kleinen Trick über die geringe Rechengenauigkeit hinweggetäuscht.

Durch „d:= r" bekommt aber die Double-Variable „d" den ungenauen Näherungswert zugewiesen - und bei der nachfolgenden Ausgabe wird dann nicht auf 6, sondern auf 15 Dezimalstellen gerundet. So führt die höhere Genauigkeit zu einem scheinbar ungenaueren Ergebnis.

Bei der Ausgabe mittels „*Write*" gibt es mehrere Optionen:

☞ `write(x)` oder `write(x:b)` oder `write(x:b:0)`

Im Bereich 10000 < x <= 0.1 wird die Zahl x „normal" ausgegeben, andernfalls in Exponentialdarstellung. In beiden Fällen werden überflüssige Nullen am Ende weggelassen.

„b" gibt hier wie immer die Breite des Feldes an, in das die Zahl rechtsbündig ausgegeben werden soll.

Bei negativen Zahlen beginnt die Ausgabe mit dem Minuszeichen, bei positiven Werten wird am Anfang ein Leerzeichen ausgegeben. Beispiele:

```
write(-4/2)                    -2
write(pi)                      3.14159265358979
write(26.731e+9:20)            2.6731E+10
```

☞ `write(x:b:n)  mit n > 0`

Die Zahl x wird in Festkommadarstellung mit n Nachkommastellen ausgegeben. Beispiele:

```
write(17.4:12:3)                17.400
write(26.731: 12: 2)            26.73
write(pi:25:25)          3.1415926535897930000000000
```

Bei „Real" sind bis zu 20 Stellen möglich. Beachten Sie aber, daß der AMIGA hier nur mit etwa 6 1/2 - stelliger Genauigkeit rechnet, so daß es schon an der 7. Stelle (Vorkommastellen mitgerechnet) zu Rundungsfehlern kommen kann und der ganze Rest nur mit Nullen aufgefüllt wird. Im „Double"-Bereich sind bis zu 25 Nachkommastellen möglich, aber auch hier werden maximal 16 „echte" Stellen ausgegeben, der Rest ist Null.

☞ `write(x:b:n)  mit n < 0`

Die Zahl x wird in Exponentialdarstellung und in „n" - stelliger Genauigkeit ausgegeben. Beispiele:

```
write(26.731:12:-4)      2.673E+01
write(0.07:12:-5)        7.0000E-02
write(0:12: -3)          0.00E+00
```

Die ganzen mathematischen Funktionen, wie „*SIN*", „*EXP*", „*SQRT*" oder „*LN*", sind nur im Real-Bereich definiert. Sie können zwar wegen der beidseitigen Wertzuweisungskompatibilität auch auf Zahlen doppelter Genauigkeit angewandt werden, liefern dann aber ungenaue Ergebnisse, nämlich in der siebenstelligen Real-Genauigkeit! In der „mathieeedoubtrans.library" finden Sie alle nötigen Funktionen in doppelter Rechengenauigkeit.

5.2 Boolean

Der Datentyp „Boolean" umfaßt die beiden Werte „True" und „False". Sie sind geordnet:

Es gilt

```
False < True
ord(False) = 0
ord(True) = 1
```

Eine Variable dieses Typs belegt ein Byte. Es gibt folgende Verknüpfungen:

`AND`	logisches Und
`OR`	logisches Oder
`XOR`	logisches auschließliches Oder

OR und XOR haben gleiche Priorität und zwar eine kleinere als AND.

Beispiel:

```
a OR b AND c XOR d
```

entspricht:

```
a OR (b AND c) XOR d
```

und damit (gleiche Priorität von OR und XOR, deshalb Auswertung von links nach rechts):

```
(a OR (b AND c)) XOR d
```

Außerdem gibt es noch die Negation NOT, die höchste Priorität hat:

```
NOT a AND b      entspricht deshalb:  (NOT a) AND b.
```

Umsteiger von BASIC (Was ist das? Nie gehört!) nach PASCAL wissen oft nicht, daß die Booleschen Verknüpfungen in PASCAL dieselbe Priorität wie arithmetische Operatoren haben. Somit wäre

```
IF x>=0 AND x<10 THEN ...
```

falsch: „AND" hat Vorrang vor den Vergleichoperatoren, so daß zuerst „0 AND x" zusammengefaßt würde (in Standardpascal nicht erlaubt, bei MaxonPASCAL ist dies, falls x eine Ganzzahl-Variable ist, eine binäre Verknüpfung, die stets den Wert 0 hat), und übrig bliebe der unsinnige und syntaktisch falsche Mehrfachvergleich

```
IF x >= (0 AND x) < 10 THEN ...
```

Richtig wäre:

```
IF (x>=0) AND (x<10) THEN ...
```

Boolean-Werte können mit *„Write"* ausgegeben werden.

Beispiel:

```
write(1=2)
```

gibt „false" aus.

5.3 Char

Ein „Char" ist ein einzelnes Zeichen („Character"). Es wird wahlweise mit einfachen oder doppelten Anführungszeichen eingeschlossen.

Folgendes wären also korrekte Char-Konstanten:

```
'A'     "A"     'x'     '\'     '2'     "'"
```

Das letzte Beispiel hätte man auch anders darstellen können: Ein Hochkomma wird in Hochkommata eingeschlossen, indem man es doppelt setzt, also '''' '. Standard-PASCAL erlaubt es nicht, ein Char mit doppelten Hochkomma einzuschließen, so daß dort die letztere die einzige zulässige Darstellung des Apostrophs wäre.

MaxonPASCAL verwendet natürlich den normalen 8-Bit-Amiga-Zeichencode. Nicht-schreibbare Zeichen sind die Codes von 0 bis 31 sowie von 128 bis 159. Dafür gibt es verschiedene Literalarten: Zum einen wäre da die gewöhnliche Schreibweise „Chr(n)", wobei „n" ein ganzzahliger Ausdruck im Bereich von 0 bis 255 ist. Falls der Parameter konstant ist, kann die Chr-Funktion auch als Konstante verwendet werden, d. h.

```
"CONST Linefeed = chr(10)" oder "CASE c OF chr(13): ..."
```

sind möglich.

Ferner akzeptiert MaxonPASCAL die von gewissen unter MS-DOS laufenden PASCAL-Implementierungen her bekannte Schreibweise „#n", wobei n eine ganzzahlige Konstante im zulässigen Bereich ist. Und last not least kann man (da Charkonstanten äußerlich Ein-Zeichen-Strings sind) wie bei Stringkonstanten verfahren, z.B. '\10 oder '$0a als Ersatz für chr(10). Näheres dazu finden Sie im String-Kapitel.

5.4 Aufzählungs- und Ausschnittstypen

PASCAL bietet Ihnen die Möglichkeit, sich Datentypen selbst zu definieren. Eine einfache Möglichkeit sind die Aufzählungstypen.

Man erzeugt diesen Datentyp ganz einfach, indem man die Werte, die jeweils durch einen Bezeichner repräsentiert werden, als Liste in runden Klammern aufzählt. Ein beliebtes Beispiel:

```
TYPE Wochentag =   (Montag, Dienstag, Mittwoch, Donnerstag,
                    Freitag, Samstag, Sonntag);
```

Diese Zeile definiert nicht nur den neuen Datentypen „Wochentag", sondern auch die sieben Konstanten „Montag" bis „Sonntag". Deshalb dürfen diese Bezeichner zuvor noch nicht definiert sein.

„Wochentag" ist ein Datentyp, der die sieben diskreten Werte „Montag" bis „Sonntag" umfaßt. „Mittwoch" z. B. ist eine Konstante mit dem Wert „Mittwoch" - klingt seltsam, ist aber so.

Welche Eigenschaften hat ein solcher Datentyp, und was kann man damit machen? Die Elemente sind geordnet: Es gilt Montag < Dienstag < Mittwoch < ... < Sonntag. Des weiteren haben sie auch Ordnungszahlen: Ord(Montag)=0, Ord(Dienstag)=1 usw. Auch die Funktionen *„Succ"* und *„Pred"* können auf Aufzählungstypen angewendet werden, z. B. Succ(Montag)=Dienstag, Pred(Freitag)=Donnerstag. Das sind dann aber auch so ziemlich alle Operationen auf Aufzählungstypen. Ihr Sinn ist im wesentlichen, Programme übersichtlicher zu gestalten, z. B. indem man Wochentage nicht durchnumeriert, sondern wie im obigen Beispiel über ihre Namen verwaltet. Eine Variable eines Aufzählungstyps belegt übrigens immer 2 Bytes.

Die andere Klasse von einfachen selbstdefinierten Datentypen sind die Ausschnittstypen. Dazu nimmt man sich einen bereits existierenden geordneten Datentypen, also einen Ganzzahltypen, Boolean, Char oder einen Aufzählungstypen und schränkt seinen Wertebereich auf ein beliebiges Interval ein. Das folgende Programmfragment enthält einige Ausschnittstypen:

```
TYPE
   Positiv     = 1..MaxInt;  { Dies ist ein Ausschnitt aus Integer,
                               der alle positiven Integerzahlen umfaßt }
   Wochentag   = (Montag, Dienstag, Mittwoch, Donnerstag, Freitag,
                  Samstag, Sonntag);  { kennen wir schon }
   Arbeitstag  = Montag..Freitag; { ein Ausschnitt aus dem Typen
                                    "Wochentag" }
   Gross       = 'A' .. 'Z'; { 26-elementiger Ausschnittstyp von
                               "Char" }
   Ziffern     = '0' .. '9'; { auch dies ist ein Char-Ausschnitt }

VAR  n:              Positiv; { die Variable "n" kann nur positive Werte
                                haben }
     i:              0 .. 2673142; { ein Ausschnitt aus "LongInt" }
     PlusMinusCard:-MaxCard .. +MaxCard; { auch dies ist ein
                                           Ausschnitt aus "LongInt" }
     Seltsam:        False .. True; { umständliche Art, eine Boolean-
                                      Variable zu deklarieren }
     Heute:          Wochentag;
     IstLeider:      Montag..Freitag;  { ist im Vergleich zu "Heute"
                                         auf 5 Werte beschränkt }
     HätteGern:      Samstag..Sonntag; { Achtung: beliebter Fehler ist,
                                         hier "(Samstag,Sonntag)" zu
                                         schreiben. Überlegen Sie doch
                                         mal, warum das nicht geht! }
```

Man definiert Ausschnittstypen also, indem man eine Unter- und eine Obergrenze angibt. Beide Grenzen müssen Konstanten sein; die untere muß kleiner als die obere sein, denn sonst meldet der Compiler einen Fehler.

Eine Variable eines Ausschnittstyps verhält sich exakt wie eine des Grundtyps. Einziger Unterschied: Bei jeder Wertzuweisung wird geprüft, ob die Variable im erlaubten Bereich liegt; andernfalls bricht das Programm mit einer Fehlermeldung ab.

5.5 Stringtypen

Ein String ist eine beliebig lange Folge von Zeichen. Es gibt zwei verschiedene Stringtypen:

- Die Deklaration „String[n]" ist identisch mit „Array [1..n] of Char". Das Symbol „String" ohne Längenangabe entspricht „String[80]". Zwar ist „String" kein Typbezeichner, sondern eine sog. Directive. Seit Version 1.5 von KICK-PASCAL kann „String" aber überall als Pseudo-Typbezeichner verwendet werden, z. B. auch als Typ in Parameterlisten.
- Der Datentyp „Str" ist ein Zeiger auf einen String.

Beim ersten Typ müssen Sie eine maximale Länge festlegen. Hier können Sie über den Index auch jedes Zeichen einzeln ansprechen. Das Ende wird mit einem Zeichen chr(0) markiert. Weil auch dieses Zeichen ein Byte benötigt, können Sie in einem „String[5]" effektiv höchstens 4 Zeichen unterbringen.

„Str" ist nichts weiter als ein Pointer. Das AMIGA-Betriebssystem benötigt häufig solche Zeiger auf Texte, und das ist auch der einzige Grund, warum dieser Typ in MaxonPASCAL implementiert wurde.

Er ist nämlich eine etwas haarige Angelegenheit. Betrachten Sie bitte einmal folgendes Programm:

```
Program strings;
Var s1:str; s2:String[10];
Begin
 s2:='Hallo';
 s1:=s2;
 s2:='Horrido';
 writeln(s1)
End.
```

Raten Sie mal, welche Ausgabe dieses Programm macht: Zuerst wird in „s2" der Text 'Hallo' abgelegt (s2[1]='H', ... , s2[5]='o', s2[6]=chr(0)). In der nächsten Zeile wird „s2" an „s1" zugewiesen.

„s1" besitzt aber keinen eigenen Speicher außer den 4 Bytes, die ein Pointer benötigt („s2" belegt 10 Bytes). Vielmehr zeigt der ZEIGER „s1" nach dieser Zeile auf die Variable „s2". Wenn dann in der nächsten Zeile ein anderer Text an „s2" zugewiesen wird, ändert sich auch der „Inhalt" von „s1". Ergo gibt das Programm 'Horrido' aus - ein eklatanter Verstoß gegen die Prinzipien gepflegter PASCAL-Programmierung.

An diesem Beispiel sehen Sie auch, daß Sie den Wert, auf den „s1" zeigt, erhalten, ohne (wie bei normalen Pointern notwendig) „s1^" zu schreiben. Das liegt daran, daß MaxonPASCAL Strings intern ohnehin nur über ihre Speicheradresse verwaltet werden.

Gefährlich wird es (und auch das dürfte es in PASCAL eigentlich nicht geben), wenn im obigen Beispiel „s2" lokal in einer Procedure und „s1" als globale Variable deklariert wäre. In der Prozedur wird „s1" zunächst ein Zeiger auf die lokale Variable „s2" zugewiesen. Wenn das Programm dann die Prozedur verläßt, existiert die Variable „s2" garnicht mehr, und „s1" zeigt deshalb irgendwo in die Wallachei. Sobald der entsprechende Speicherbereich wieder benutzt wird, wird der Wert von „s1" geändert.

Dieser Exkurs zeigt, daß Sie den Str-Typ hauptsächlich im Zusammenhang mit der AMIGA-Betriebssystemprogrammierung verwenden sollten. Er ist auch ohnehin nicht so interessant, weil man eine solche Variable weder über Read/ReadLn einlesen noch die Zeichen einzeln ansprechen kann. Alles, was Sie mit „str" anfangen können, ist:

- eine Variable zuweisen.
- eine Stringkonstante zuweisen, z.B. "s1:='Halleluja'". Eine solche Zuweisung kann bedenkenlos auch in einer Prozedur niedrigeren Ranges erfolgen, denn die Konstante liegt nicht im Variablenspeicher, sondern im Programmcode.
- mit anderen Strings vergleichen.
- mit Write(ln) ausgeben.
- als Parameter in Funktionsaufrufen verwenden.

Damit wären wir auch schon beim nächsten Thema:Operationen auf Strings.

MaxonPASCAL hat folgendes auf Lager:

- Wertzuweisungen - wie oben gesehen.
- Ausgabe mit Write/WriteLn.
- Eingabe mit Read/ReadLn (geht nicht mit Str).
- Vergleichsoperationen =, <>, <, <=, >, >=. Es kann auch ein String mit einem Char verglichen werden.
- Verbinden mit „+" oder der „*Concat*" - Funktion - Mit „*Copy*" auf Ausschnitte zugreifen.

Stringkonstanten können wahlweise mit ' ' (wie in Standard-PASCAL) oder mit " " eingeschlossen werden. Es gibt die Möglichkeit, Steuerzeichen in Strings einzusetzen: Man unterbricht den String mit ' oder „ - aber mit dem selben Zeichen, mit dem man ihn begonnen hat - und kann dann entweder mit $xx eine Hexadezimalzahl oder mit \ eine Dezimalzahl einschieben.

Beispiel:

```
Program Steuercodes;
Begin
  writeln('xxx aaa'$0d'bbb');
  { $0d oder dezimal 13 entspricht dem Code "CR" ("Wagenrücklauf").
  Der Cursor springt an dieser Stelle an den Zeilenanfang, so daß
  "xxx" von "bbb" überschrieben wird. }
```

```
writeln("Zeile 1"\10$a"Zeile 2");
{ Dezimal 10 = Hex $a entspricht "LF" ("Linefeed"/"Zeilenvor-
schub"). Hier werden zwei Zeilenenden eingeschoben. }

writeln(''$9b'33mJetzt '\155'32;4mwird''s'\155'31;0;3m bunt!''$0a);
{ Der Steuercode dez 155 bzw. Hex $9b heißt "Escape" oder "CSI"
("Control Sequence Introducer" - "Steuersequenzeinleiter"). Damit
haben Sie enorm viele Möglichkeiten, die ich hier nicht einzeln
aufzählen kann. Setzt man z. B. dahinter eine oder mehrere
Dezimalzahlen (ggf. durch ";" getrennt) und beschließt die Sequenz
mit einem kleinen "m", wird das Aussehen des Textes beeinflußt: die
Zahlen 0..7 stellen verschiedene Textattribute ein (unterstrichen,
kursiv, ...) und Zahlen 30..37 setzen die Schriftfarbe.
Außerdem fällt auf, daß
a) der String mit einem Steuercode beginnt, indem der Leerstring ''
   vorangestellt wird,
b) ein Apostroph ' im String enthalten ist, indem er doppelt
   gesetzt wird und
c) die Zeichenkette mit einem Steuerzeichen endet, ohne daß
   dahinter noch ein Hochkomma stehen muß. }

writeln('Zeile 4'\n'Zeile'\e'33;0m 5');
{ Hinter "\" können Sie auch die Buchstaben "n" und "e" setzen.
Ersterer entspricht dem Zeichen LF (Code $0a), letzterer "CSI"
(Code $9b). }

End.
```

Problematisch wird es, wenn auf oben beschriebene Weise ein Nullbyte in einen String eingesetzt wird. Sowohl MaxonPASCAL als auch das AMIGA-Betriebssystem betrachten dies in der Regel als Stringendemarkierung, so daß der Rest eventuell verloren geht. Wenn Sie einmal eine solche Null benötigen - z.B. bei der Intuition-Funktion *„DisplayAlert"* - müssen Sie den String äußerst vorsichtig behandeln.

Wenn ein String mit einem Steuerzeichen beginnen soll, können Sie am Anfang statt des Leerstrings auch das Zeichen „#" verwenden. Außerdem kann „#" überall als Ersatz für den Backslash „\" verwendet werden. Eine Zeile aus dem obigen Beispiel könnte also auch so aussehen:

```
write(#$9b'33mJetzt '#155'32;4mwird''s'#155'31;0;3m bunt!''$0a);
```

Durch solche Steuercodes kann ein String ganz schön lang werden. Er darf aber normalerweise das Zeilenende nicht überschreiten. Zu diesem Zweck gibt es die Möglichkeit, die beiden Zeichen „\&" in den String einzuschieben. Dahinter können dann beliebig viele Zeilenenden, Leerzeichen oder sogar Kommentare stehen, bevor der String fortgesetzt wird.

Beispiel:

```
write('Ein Gedicht:'\n\n\e'32mvon Douglas Adams'\n\n\&
      \x'[33mO zerfrettelter Grunzwanzling!'\n\&
      'Dein Harngedränge ist für mich'\n\&
```

```
'Wie Schnatterfleck auf Bienenstich.'\&
\n\n\c'31mDen Rest erspare ich Ihnen.')
```

Dieser Befehl gibt einen mehrere Zeilen langen Text auf einen Schlag aus. Allerdings ist in MaxonPASCAL die Länge von String-Konstanten auf 400 Zeichen beschränkt - um einen Roman auszugeben, werden Sie also doch mehr als eine Zeichenkette benötigen.

Im obigen Beispiel tauchen außerdem die Einschübe „\c" und „\x" auf.

„\x" entspricht dem Escape-Zeichen #$1b. Zusammen mit einer folgenden eckigen Klammer „[„ ist es äquivalent zum CSI-Zeichen. Diese Folge hat aber den Vorteil, daß sie auch vom Druckertreiber verstanden wird. Man kann \x'[' mit \c abkürzen.

Es gibt noch zwei wichtige Funktionen zur Stringbehandlung: „Concat" und „Copy":

```
Concat(string1, string2, ...)
```

hängt die übergebenen Strings zusammen. Das Ergebnis wird als „temporärer String" zurückgegeben.

„Was ist das?" Die Variablen eines „String[N]"-Typs besitzen jeweils einen für sie reservierten (N Bytes langen) Speicherbereich. Auch eine Stringkonstante liegt an einer bestimmten Stelle im Speicher, und ein „Str"-Zeiger zeigt wenigstens auf eine Speicheradresse. Das Ergebnis, das „Concat" zurückgibt, hat aber keinen vorher bestimmten Speicherplatz, vielmehr wird unmittelbar vor dem Zusammenhängen irgendwo die benötigte Menge Speicher reserviert (denn das Ergebnis kann ja beliebig lang werden) und muß nachher wieder freigegeben werden. Der String ist also quasi nach dem Lesen zu vernichten.

(Wußten Sie schon: Dokumente der höchsten NATO-Geheimhaltungsstufe tragen den Stempel „DBR" - „Destroy Before Reading"....) Von der internen Verwaltung dieser Strings merken Sie als PASCAL-Programmierer in der Regel nichts, denn wenn Sie einen „Concatinierten" String mit Write ausgeben oder an eine Stringvariable zuweisen, managed das PASCAL-Laufzeitsystem das Ganze für Sie.

Eine Einschränkung gibt es aber: Ein temporärer String kann nicht an eine „Str"-Variable zugewiesen werden. Grund: Ein temporärer String ist ja nur „von begrenzter Lebensdauer", und eine Str-Variable hat ja auch keinen Speicher, in dem sie den String aufbewahren könnte, sondern kann lediglich auf eine Zeichenfolge zeigen. Und ein Zeiger auf einen String, der sich bald in Wohlgefallen auflöst, wäre ja wenig sinnvoll.

Deshalb ist es natürlich auch nicht erlaubt, solche Strings als Parameter an Prozeduren und Funktionen zu übergeben, die einen „Str"-Parameter erwarten. Dazu gehören z. B. die PASCAL-Standardprozedur „Assign(datei, namestr)", die Funktion „Pos(str1, str2)" und jede Menge AMIGA-Systemfunktionen. Sie müssen hier also bei Bedarf das „Concat"-Ergebnis zunächst an eine Stringvariable zuweisen. Übrigens: Sie können Strings ersatzweise auch mit „+" aneinanderhängen.

Beispiel:

```
s := 'Hallo'+'Du'+Concat(Concat('Alter','Eimer','Wie)+'Geht's', '?')
```

Danach hat s den Wert „HalloDuAlterEimerWieGeht's?"

```
Copy(string, pos, len)
```

Diese Funktion liefert einen „len" Zeichen langen Ausschnitt ab Position „pos" aus der Zeichenkette „string". Auch hier ist das Ergebnis ein temporärer String, es gilt also sinngemäß dasselbe wie bei der Konkatination (lachen Sie nicht, dieses Wort gibt's wirklich).

Beispiel:

```
write(Copy('PASCAL the best - .... the rest!',8,3))
```

gibt „the" aus.

Wahrscheinlich war die erste Programmiersprache, die Sie gelernt haben, BASIC, so daß Sie es wahrscheinlich gewohnt sind, mit „mid$(a$,17,1)" auf das 17te Zeichen von „a$" zuzugreifen. Trotzdem sollten Sie (so schwer das auch oft fällt) mit dieser schlechten Gewohnheit brechen und stets a[17] statt Copy(a,17,1)) schreiben (Vorausgesetzt, a ist eine Variable eines String[x]-Typs). Dafür gibt es gleich vier gute Gründe:

1. Ersteres ist kürzer.
2. Es geht deutlich schneller.
3. Außerdem ist es Standard-PASCAL-kompatibel.
4. „Copy" gibt stets einen String zurück, so daß Sie das Ergebnis nicht an eine Char-Variable zuweisen können, auch wenn es nur ein Zeichen lang ist.

Bei einem Stringausdruck, einer Zeichenkettenkonstanten oder einer „Str"-Variablen ist es nicht möglich, wie oben beschrieben, auf ein einzelnes Zeichen zuzugreifen. Dafür gibt es - als besonderes MaxonPASCAL-Feature ab Version 1.5 - eine andere Schreibweise: hinter jeden Zeichenkettenausdruck kann ein Punkt, gefolgt von einem Pseudo-Index, gesetzt werden.

Ein Beispiel:

```
Program streawkceuR; Var Vorname, Nachname: String;
  i: integer;
Begin
  write('Vorname:  '); readln(Vorname);
  write('Nachname: '); readln(Nachname);
  For i:=12 Downto 1 Do
    write('Ihr Name ist'.[i]);
  writeln(':')
  For i:=Length(Vorname)+Length(Nachname)+1 Downto 1 Do
    write((Vorname+' '+Nachname).[i])
End.
```

Dieses Prográmmchen fragt den User nach seinem Namen und gibt dann erst den Text „Ihr Name ist" und dann den vollständigen Namen rückwärts aus, indem es die beiden Stringausdrücke zeichenweise von hinten nach vorn schreibt.

Was wäre in jenem Programm der Unterschied zwischen „Vorname[5]" und „Vorname.[5]"? Ersteres ist eine Char-Variable, nämlich das fünfte Element des Char-Arrays „Vorname". Dieser Variablen kann man z. B. auch einen Wert zuweisen. Das zweite ist ein Char-Ausdruck, so wie „Chr(k+8)" oder „Succ('A')", und kann deshalb nicht auf der linken Seite einer Wertzuweisung auftreten. „Stringausdruck.[Index]" ist also identisch mit „Copy(Stringausdruck,Index,1)", nur daß ersteres ein Char und letzteres einen String liefert.

5.6 Pointertypen

Außer den normalen Pointern besitzt MaxonPASCAL noch einen typfreien Pointertypen namens „ptr". Eine Variable dieses Typs kann auf einen beliebigen Typ zeigen - weshalb es auch nicht möglich ist, mit „ ^ " diese Variable anzusprechen. Natürlich kann auf einen solchen Zeiger auch nicht New oder Dispose angewandt werden, denn es ist ja gar nicht klar, wie viel Speicher dabei reserviert bzw. freigegeben werden muß.

Wahrscheinlich werden Sie sich jetzt fragen, welchen Sinn dann ein solcher Zeiger hat. Nun, im Gegensatz zu normalen Zeigern kann an eine Ptr-Variable ein beliebiger Pointertyp zugewiesen werden und umgekehrt. Ein Beispiel:

```
Program Pointer;
  Var p1:^Typ1; p2:^Typ2; p:ptr;
  Begin
    ... p:=p1; p2:=p; ...
```

Das Ergebnis dieser Zuweisungen: p2 zeigt auf dieselbe Variable wie p1, obwohl diese eigentlich verschiedene Typen haben (im nächsten Abschnitt werden Sie erfahren, wie so etwas auch einfacher geht).

Jetzt werden Sie wahrscheinlich denken: OK, aber welchen Sinn hat SO ETWAS denn nun wieder? Darauf sei ebenso kurz wie ausweichend geantwortet, daß Sie das zur System-Programmierung benötigen, worauf in einem der nächsten Kapitel noch eingegangen wird.

NEW und *DISPOSE* verwenden übrigens nicht den reservierten Arbeitsspeicher, sondern die Exec-Funktionen *„Allocmem"* und *„Freemem"*, falls Ihnen das etwas sagt (falls nicht: erstere fordert: „Bitte ein Kilo Byte!" vom Betriebssystem Speicher an, letztere gibt ihn wieder zurück). Somit schränken dynamische Variablen nicht Ihren Stack-Speicher ein. Es wird aber bei *„New"* quasi Buch geführt, welche Speicherbereiche reserviert wurden. Dadurch kann es zum einen bei *„Dispose"* nicht zu einem Absturz kommen (Guru Meditation Number $81000009), und zum anderen wird der Speicher am Programmende wieder ordentlich freigegeben

Und noch eine Besonderheit hat MaxonPASCAL im Zusammenhang mit Pointern auf Lager: Wenn Sie einer beliebigen Variablen ein „ ^ " voranstellen, erhalten Sie einen Zeiger auf diese Variable.

Beispiel:

```
Program Pointer2;
  Var i:integer; p:^integer;
  Begin
    i:=26731; p:=^i; writeln(p^)
  End.
```

Ein anderes Beispiel: „MyScreen" sei der Zeiger, den Ihnen die Intuition-Funktion „*OpenScreen*" (Typ: ^ Screen) zurückgegeben hat.

Mit der Prozedur „*ClearScreen*" aus der Graphics-library wollen Sie nun diesen Bildschirm löschen. Dazu brauchen Sie einen Zeiger auf den sog. Rastport des Screens, der als Feld (also nicht in Form eines Pointers!) in der Screenstruktur (Struktur=Record) enthalten ist. Der Befehl sieht so aus:

```
ClearScreen(^MyScreen^.Rastport);
```

Die Schreibweise mit den zwei „ ^ " ist zugegebenermaßen etwas ungewohnt. Hier sehen Sie auch, daß die Auswertung quasi von rechts nach links erfolgt: Das erste „ ^ " bezieht sich auf den ganzen Ausdruck „MyScreen ^ .Rastport" und nicht nur auf „MyScreen" (zumal ein Zeiger auf einen Zeiger keine allzu nützliche Sache ist).

Natürlich sollten Sie sich hüten, einen Pointer zu DISPOSEn, den Sie mit „ ^ " auf eine NICHT durch New erzeugte Variable gesetzt haben. Die Folge wäre aber nur eine Fehlermeldung, denn MaxonPASCAL paßt hier auf.

5.7 SET-Typen

Sets können ziemlich groß werden: der Elementbereich darf bis zu 2 ^ 16 - 16 = 65520 Werte umfassen, wobei stets die untere Grenze zur nächsten durch 16 teibaren Zahl abgerundet und die obere zur nächsten höheren (!) durch 16 teilbaren Zahl aufgerundet wird. So ist z. B.

```
VAR s: SET of 0..65519;
```

eine der größtmöglichen Mengen im LongInt-Bereich. „1..65520" geht nicht, denn hier wird die 1 auf 0 ab- und die 65520 auf 65536 aufgerundet. Im Integerbereich wäre entsprechend

```
VAR t:set of -32752..32767;
```

gerade noch möglich - also fast schon „Set of integer"!

Elementtypen können alle geordneten Datentypen sein, also Char, Boolean, Ganzzahl- und Aufzählungstypen sowie Ausschnitte daraus.

Insgesamt gibt es folgende Verknüpfungen auf Mengen:

*	Schnittmenge:	[1..8]*[5..10] = [5..8]
+	Vereinigung:	[1,2] + [2,6..10] = [1,2,6..10]
-	Differenz:	[-10..10] - [-Maxint..0] = [1..10]
=	Mengenvergleich:	[17,4,21] = [4,17,21] ist TRUE.
<>	Gegenteil von =	

<=	Teilmenge:	[1,7] <= [0..10]	ist TRUE.
>=	genau andersrum:	[0..10] >= [1.7]	ist TRUE.
IN	„ist Element von“:	5 IN [1..10]	ist TRUE.

Auch hier gelten die normalen „Punkt- vor Strichrechnung“-Regeln:

```
x IN a-b*c      entspricht      x IN (a-(b*c))
```

Sie sollten aber immer bedenken, daß Mengenoperationen in MaxonPASCAL (wie in wohl fast allen anderen PASCAL-Implementierungen) nicht gerade schnell sind. Genaugenommen sind sie sogar total lahm, vor allem dann, wenn die beteiligten Mengen groß werden. Deshalb sollten Mengen solchen Verwendungszwecken vorbehalten bleiben, bei denen es nicht auf Geschwindigkeit ankommt.

Eine Ausnahme gibt es aber: Falls hinter „IN“ eine Mengenaufzählung steht, die ausschließlich aus Konstanten besteht, wird dies von MaxonPASCAL zu einer Folge von Vergleichsoperationen optimiert. So ist z. B.

```
L in [0,10..20,100..120]
```

erheblich schneller als

```
(L=0) or (L>=10)and(L<=20) or (L>=100)and(L<=120).
```

5.8 Strukturierte Datentypen: Arrays und Records

Strukturierte Variablen sind allgemein Zusammenfassungen mehrerer Variablen zu einer größeren Einheit. Es gibt in PASCAL zwei Arten von solchen Variablen: Feldtypen („Arrays“) und Verbundtypen („Records“).

Betrachten wir zuerst die Felder. Ein Feldtyp wird definiert durch

```
"ARRAY" "[" Indextyp "]" "OF" Elementtyp
```

Der Indextyp kann ein beliebiger geordneter Datentyp sein, also ein Ganzzahltyp, Char, Boolean, ein Aufzählungstyp oder ein Ausschnitt aus einem dieser Typen. In der Regel benutzt man hier aber Ausschnittstypen. Der Elementtyp kann ein beliebiger Datentyp sein.

Es wird jedem Wert des Indextyps eine Variable des Elementtyps zugeordnet.

Ein Beispiel:

```
VAR Feld: ARRAY[1..10] OF F Char;
```

Die Variable „Feld“ besteht aus den zehn Elementen „Feld[1]“ bis „Feld[10]“, die jeweils Char-Variablen sind. Das Ganze ist übrigens identisch mit dem Datentypen „String[10]“ (siehe Abschnitt 5 dieses Kapitels).

Alternativ hätte man auch so deklarieren können:

```
TYPE FeldIndex = 1..10;
     FeldTyp = ARRAY[FeldIndex] OF Char;
VAR  Feld: FeldTyp;
```

Der Elementtyp kann ein beliebiger Datentyp sein - also auch ein weiterer Array-Typ, z. B.

```
TYPE DreiD = ARRAY[-1..1] OF
             ARRAY['a'..'z'] OF
             ARRAY[Boolean] OF
             integer;
```

Das kann abgekürzt werden durch:

```
TYPE DreiD = ARRAY[-1..1, 'a'..'z', Boolean] OF integer;
```

Entsprechend kann man, wenn „Arr" eine Variable des Typs „DreiD" ist, ein Element wahlweise mit

```
Arr[ i+1 ] [ 'x' ] [ i=j ]
```

oder

```
Arr[ i+1, 'x', i=j ]
```

ansprechen.

Die Syntax für „Array-Typ" ist also:

```
"ARRAY" "[" GeordneterTyp { "," GeordneterTyp } "]" "OF"s Typ
```

Die Anwendungsmöglichkeiten von Arrays sind so vielfältig, daß jeder, der überhaupt schon einmal in irgendeiner Sprache programmiert hat, auf Anhieb zahlreiche Beispiele dafür aus dem Schatz seiner Erfahrungen nennen kann. Deshalb will ich Sie hier nicht mit einem Beispielprogramm dafür langweilen, sondern gleich zum zweiten strukturierten Datentyp übergehen: dem Record.

Ein Record- oder Verbundtyp faßt mehrere Variablen beliebigen, auch unterschiedlichen, Typs zu einer Verbundvariablen zusammen. Ein klassisches Beispiel aus der Mathematik: eine komplexe Zahl besteht aus einem Realteil und einem Imaginärteil, die jeweils reelle Zahlen sind. In PASCAL sähe das so aus:

```
TYPE Komplex = RECORD
                 Re: Real;
                 Im: Real
               END;
```

Das kann abgekürzt werden zu:

```
TYPE Komplex = RECORD
                 Re, Im: Real
               END;
```

Eine Variable „x" des Typs „Komplex" besteht dann aus zwei „Feldern", die „x.Re" und „x.Im" heißen und jeweils Real-Variablen sind.

Ein anderes Beispiel: Von 10000 Studenten sollen Vorname, Nachname (jeweils Strings mit bis zu 40 Zeichen) und Matrikelnummer (LongInt)) gespeichert werden.

Dies läßt sich realisieren, indem man drei Arrays deklariert: Zwei Stringarrays für die Vor- bzw. Nachnamen und ein „ARRAY OF Long" für die Matrikelnummern. Aber sinngemäß gehören ja nicht alle Vornamen zusammen, sondern jeweils der Vorname, der Nachname und die Nummer jedes einzelnen Studenten. Um diese Zusammengehörigkeit im Programm klarzumachen, bietet sich die Verwendung von Records an:

```
VAR   Studies = ARRAY [1..10000] OF RECORD
                                      Vorname, Nachname: String[40];
                                      Matrikelnummer: Long
                                    END;
```

Eine Zuweisung an den k-ten Studenten könnte dann so aussehen:

```
Studies[k].Vorname:= 'Willi';
Studies[k].Nachname:= 'Wacker';
Studies[k].Matrikelnummer:= 3710815;
```

Einige Angaben zur Syntax:

```
Record-Typ:       "RECORD" Feldliste "END"

Feldliste:        { Bezeichner { "," Bezeichner } ":" Typ [ ";" ] }
                  [ Variantenteil ]

Variantenteil:    "CASE" [ Bezeichner ":" ] Typ-Bezeichner "OF"
                  { Konstante { "," Konstante } ":" "(" Feldliste ")"
                  [ ";" ] }
```

Bevor ich den „Variantenteil" erläutere, möchte ich darauf hinweisen, daß die Feld-Bezeichner einer jeden Feldliste einen eigenen Gültigkeitsbereich haben. Dies bedeutet, daß verscheidene Record-Typen gleichnamige Felder haben können und daß ein Feld einen Namen haben kann, der bereits vergeben ist. Einzige Bedingung ist, daß innerhalb eines einzelnen Records keine zwei Felder den gleichen Bezeichner tragen dürfen.

Oft kommt es vor, daß man in einem Recordtypen Daten speichern will, die sich gering unterscheiden. Beispiel: Es soll der Warenbestand eines Kaufhauses verwaltet werden. Wir begnügen uns zunächst mit der Warenbezeichnung, dem Verkaufspreis und einer Warengruppe. Bei Lebensmitteln wollen wir aber zusätzlich das Verfallsdatum und bei Kleidung die Konfektionsgröße speichern. Der erste Versuch mit einem normalen Record und einem Aufzählungstypen für die Kategorie könnte so aussehen:

```
TYPE Warengruppe = (Lebensmittel, Kleidung, Sonstiges);
     Ware = RECORD
       Bezeichnung: STRING[50];
```

```
    Preis: RECORD    { Schachtelung von Record-Typen! }
             DM: integer;   { Mark }
             Pf: 0..99      { Pfennige }
           END;
    Kategorie: Warengruppe;
    Verfall:  RECORD
                Tag, Monat, Jahr: integer
              END;
    Konfgroesse: integer;
  END;
```

Der Haken an der Sache: Wir benutzen immer entweder das Verfallsdatum oder die Größe oder keins von beiden und verschwenden deshalb stets mindestens ein Feld. Das können wir mit einem Varianten-Teil ändern:

```
Ware = RECORD
         Bezeichnung: STRING[50];
         Preis: RECORD    { Schachtelung von Record-Typen! }
                  DM: integer;   { Mark }
                  Pf: 0..99      { Pfennige }
                END;
         CASE Kategorie: Warengruppe OF
           Lebensmittel: (Tag, Monat, Jahr: integer);
           Kleidung:     (Konfgroesse: LongInt);
           Sonstiges:    ( );
       END;
```

Die CASE-Zeile bewirkt zunächst einmal, daß ein Feld „Kategorie" des Typs „Warengruppe" eingerichtet wird. Der ganze Rest der Feldliste ist dann ein Varianten-Teil. Jede Variante beginnt mit einer Liste von Konstanten (im Beispiel bestehen die „Listen" jeweils nur aus einer Konstanten), gefolgt von einem Doppelpunkt - also ähnlich wie beim CASE-Statement. Dahinter folgt jeweils eine Feldliste - aber diesmal nicht mit „RECORD" ... „END", sondern von runden Klammern umschlossen.

Die Feldliste kann übrigens auch leer sein, wie Sie in der Zeile „Sonstiges" sehen.

Was hat das dann zu bedeuten? Die Felder „Tag", „Monat" und „Jahr" belegen den selben Speicher wie das Feld „Konfgroesse". Die Zeile für den Fall „Sonstiges" ist vollkommen überflüssig, fördert aber die Lesbarkeit, weil dadurch klar wird, daß das Feld „Kategorie" auch diesen Wert annehmen können soll. Als Beispiel dafür, wie man mit solchen Records arbeitet, hier eine Prozedur zur Warenausgabe:

```
PROCEDURE Ausgabe(W: Ware);
  BEGIN
    writeln(W.Bezeichnung);
    writeln(W.Preis.DM , '.' , W.Preis.Pf);
    IF Kategorie=Lebensmittel THEN
      writeln('Haltbar bis: ', W.Tag, '.' , W.Monat, '.' , W.Jahr)
    ELSE
      IF Kategorie=Kleidung THEN writeln('Größe ', Konfgroesse )
  END;
```

Wie Sie sehen, sind die Felder des Variantenteils ganz normale Record-Felder, nur daß sie sich zum Teil überlappen. Eigentlich sollte es zu einer Laufzeit-Fehlermeldung führen, wenn man auf „Tag" zugreift, obwohl „Kategorie" den Wert „Kleidung" hat. MaxonPASCAL prüft dies aber nicht und überläßt es so Ihrer Verantwortung.

Der Variantenteil sieht zwar auf den ersten Blick wie ein CASE-Befehl aus, aber hier ist kein ELSE-Teil erlaubt. Auch können keine Ausschnitte wie „Konst1..Konst2" von Konstanten angegeben werden. Jeder Record kann höchstens einen Variantenteil haben. Falls Sie mehr benötigen, können Sie das über geschachtelte Records erreichen.

Beispiel: In unser Record soll noch die bevorratete Menge aufgenommem werden, entweder in Kilogramm, Litern (Real-Zahl) oder in diversen Stückzahlen (LongInt).

```
TYPE
  Warengruppe = (Lebensmittel, Kleidung, Sonstiges);
  Mass = (Masse, Volumen, Flaschen, Kisten, Stueck);

  Ware = RECORD
           Bezeichnung: STRING[50];
           Preis: RECORD
                    DM: integer;
                    Pf: 0..99
                  END;
           Menge: RECORD
                  CASE Einheit: Mass OF
                    Masse:                    (Kilogramm: Real);
                    Volumen:                  (Liter: Real);
                      Flaschen, Kisten, Stueck: (Anzahl: LongInt);
                  END;
           CASE Kategorie: Warengruppe OF
             Lebensmittel: (Tag, Monat, Jahr: integer);
             Kleidung:       Konfgroesse: LongInt);
  { Hier wurde die überflüssige Zeile weggelassen }
         END;
```

Der Mengen-Record besteht auschließlich aus einem Variantenteil (außer dem Feld „Einheit", das es immer gibt). In unsere Prozedur „Ausgabe" können wir nun an geeigneter Stelle folgendes einfügen:

```
CASE W.Menge.Einheit OF
  Masse:    writeln(W.Menge.kilogramm , 'kg');
  Volumen:  writeln(W.Menge.Liter, 'Liter');
  Flaschen: writeln(W.Menge.Anzahl, 'Flaschen');
  Kisten:   writeln(W.Menge.Anzahl, 'Kisten');
  Stueck:   writeln(W.Menge.Anzahl, 'Stück');
END;
```

Das Ganze hätte man mit einer WITH-Anweisung abkürzen können, aber das soll jetzt nicht unser Thema sein. Statt dessen noch einige Anmerkungen zu „Variant parts": das Feld hinter dem „CASE" „ kann nicht nur - wie in unseren Beispielen - als Aufzählungstyp, sondern als beliebiger geordneter Typ definiert werden, d. h. Boolean, ganzzahlig oder Char. In der Regel drängt sich die Verwendung

eines Aufzählungstyps aber geradezu auf. Die Konstanten der Varianten müssen aber zu diesem Datentyp passen. Es ist nicht nötig, für alle denkbaren Werte eine Variante anzugeben, z. B. wurde im obigen Beispiel an Schluß die Zeile für „Sonstiges" weggelassen.

Es ist auch nicht erforderlich, ein Feld zum Abspeichern der Variante einzurichten. Wenn aus irgendeinem anderen Zusammenhang hervorgeht, welcher Zweig benutzt wird, reicht es, hinter dem „CASE" nur einen Typ-Bezeichner anzugeben.

Beispiel:

```
TYPE
  Komisch = RECORD
              Inhalt: STRING[10];
              Nummer: LongInt;
              CASE integer OF
                0, 1:  (Flag: Boolean;
                        Operand1, Operand2: Real);
                2, 3,  4: (Parameter: Real);
                       5: (Fehlercode: integer;
                           CASE Boolean OF
                             true: (Re, Im: Real);
                             false: (Wert: Double)
                           )
                       -1: ()
            END;
```

Hier haben wir auch ein Beispiel für Felddeskriptoren der Typen „Integer" und „Boolean" und für geschachtelte Varianten-Teile.

Eine Record-Variable mit Variantenteil belegt stets so viel Speicher, wie die längste Variante benötigt. Der Jensen-Wirth-Standard kennt für dynamische (also über Pointer referierte) Record-Variablen mit Variant Part eine besondere Version der „NEW"-Anweisung, bei der die gewünschte Variante angegeben wird. Wäre z. B. „P" ein Pointer auf den oben definierten Datentypen „Komisch", so würde

```
New (P, 5, false)
```

für die dynamische Variable „P^" genau so viel Speicher allozieren, wie die Variante „5" mit Unterrvariante „false" belegen würde,

```
New (P, -1)
```

würde entsprechend Platz für eine Variable mit leerem Variantenteil machen usw. Ich verwende bewußt den Konjunktiv, denn MaxonPASCAL unterstützt dieses Feature nicht. Sie vergeuden also evtl. etwas Speicherplatz.

5.9 Typkonvertierungen

MaxonPASCAL „managed" im allgemeinen die Kombination unterschiedlicher Datentypen selbstständig, so daß Sie sich darüber keine Gedanken zu machen brauchen. Der Compiler konvertiert Datentypen, wenn es sinnvoll ist (z.B. Integer ung LongInt) und meldet Fehler, wenn die Typen nicht zusammenpassen, z.B. Real und Pointer. Hin und wieder kommt es aber vor, daß Sie dem Compiler Ihren Willen aufzwingen müssen.

Beispiel:

```
Program IntToLong;
Var a,b:integer;
Begin a:=26731; b:=20000; writeln(a+2*b) End.
```

Da beide Variablen vom Typ „Integer" sind, rechnet das Programm in diesem Wertebereich - und es kommt zu einem Überlauf, denn der größte in „Integer" darstellbare Wert ist MaxInt=32767. Im Bereich „LongInt" könnte der Ausdruck aber korrekt ausgewertet werden.

Für solche (und andere) Zwecke gibt es in MaxonPASCAL die Möglichkeit zur Typumwandlung. Dazu verwendet man den Bezeichner des gewünschten Typs (hier: „LongInt" bzw. „Long") wie eine Funktion.

Beispiel:

```
... writeln(Long(a+2*b)) ...
```

Dies ist zwar syntaktisch richtig, bringt aber noch keine Verbesserung unseres kleinen Programms, denn der Ausdruck wird nach wie vor zunächst als „Integer" ausgewertet und erst danach in „Long" umgewandelt. Wenn Sie den Compiler dazu zwingen wollen, tatsächlich in „Long" zu rechnen, müssen Sie dafür sorgen, daß die Bestandteile des Ausdrucks vom Typ „Long" sind:

```
... writeln(Long(a)+Long(2)*Long(b)) ....
```

Dieser Ausdruck wird korrekt ausgewertet, ist aber etwas umständlicher als nötig. Es reicht ja schließlich, wenn ein Term „lang" ist, denn dann wird der Rest automatisch umgewandelt. Aber Vorsicht:

```
... writeln(Long(a)+2*b) ...
```

wäre falsch, denn (Punktrechnung vor Strichrechnung!) der Teil- ausdruck „2*b" wird zuerst in „Integer" ausgewertet und dann erst nach „Long" konvertiert! Richtig wäre dagegen:

```
writeln(a+Long(2)*b)  oder  writeln(a+2*Long(b)),
```

denn nun wird die Multiplikation korrekt als „LongInt" ausgeführt und dann die Addition ebenfalls, denn der zweite Summand ist ja lang.

Das klingt jetzt wahrscheinlich alles etwas verwirrend, aber in der Praxis werden Sie mit solchen speziellen Fällen wohl nur selten zu tun haben. Die Moral aus der Geschicht' ist aber, daß Sie im Zweifelsfall dem Compiler bei der Typkonvertierung nicht zuviel Freiheit lassen sollten.

Was wir hier mit „Long" gemacht haben, geht auch mit den vier anderen Ganzzahltypen. Sie können damit - falls das einmal sinnvoll sein sollte - Zahlen sogar „verkürzen", also bleistiftsweise von „Long" nach „Integer" oder von „Cardinal" nach „Byte".

Noch ein Beispiel:

```
Program Kardinal;
Var c:Cardinal;
Begin c:= 2*25000; writeln(c) End.
```

Falls Sie die Compiler-Option „Arithmetic Overflow" angewählt haben, wird ein Überlauf gemeldet, denn die hier auftretenden Zahlen 2 und 25000 liegen beide sowohl im Integer- als auch im Cardinalbereich.

In diesem Fall wählt der Compiler als Rechenbereich „Integer" (den Kontext, daß als Ergebnis eine Cardinalzahl zwecks Zuweisung an eine solche Variable gewünscht wird, beachtet er nicht). Auch hier können Sie ihn zum Rechnen in „Cardinal" zwingen, in dem Sie eine („Cardinal(2)*25000" bzw. „2*Word(25000)") oder beide Zahlen („Word(2)*Word(25000)") mit „Word" oder „Cardinal" zwangskonvertieren.

Bei all' diesen Beispielen ging es nur darum, einen korrekten Programmablauf zu gewährleisten. Es gibt aber auch die Möglichkeit, mit Typumwandlungen der Syntax ein Schnippchen zu schlagen, nämlich bei den Pointern. Denn auch Pointertypen und der Stringpointer „str" können untereinander umgewandelt werden. Außerdem können sie in LongInt-Zahlen und umgekehrt Zahlen in Zeiger verwandelt werden.

Beispiel:

```
... Var p:^Typ; ... New(p); writeln(Long(p)); ...
```

Dieses Fragment gibt die Adresse (als Zahl!) aus, auf die der Pointer „p" nach dem „New" zeigt.

Noch 'n Beispiel:

```
Program Speicherzugriff;
Var p:^Word;
    w:Word;
Begin
  p:=ptr($dff180);
  For w:=0 To MaxWord Do p^:=w
End.
```

Hier wird der Pointer „p" ganz gezielt auf die Adresse $dff180 gesetzt, wo nichts anderes als die Hintergrungfarbe des Bildschirms liegt. In der For-Schleife wird dann diese Farbe schnell geändert. Ergebnis: Ein Flimmern, an dem Ihr Augenarzt seine Freude haben würde.

Achtung: Setzen Sie einen Zeiger (fast) nie auf eine ungerade Adresse! Der Prozessor MC 68000 (steckt in Ihrem AMIGA drin - es sei denn, Sie besitzen ein 680x0-Turbo-Board mit x=2,3,4,5,.. oder einen Amiga 1200, 2500, 3000, 4000,...) kann 16- und 32-Bit-Zahlen nur von geraden Adressen lesen bzw. dahin schreiben. Bei ungeraden Adressen stürzt er auf einfachen 68000 Prozessoren ab (mit der Guru Meditation Number $00000003.xxxxxx). Dies sollten Sie also, wie gesagt, fast nie tun. „Fast" deshalb, weil 8-Bit-Daten (Byte, ShortInt, Boolean, Char) durchaus an ungeraden Adressen liegen dürfen.

Es gibt aber noch viele andere Möglichkeiten zur Typwandlung. Da wären zum einen die geordneten Datentypen, also Boolean, Char, Ganzzahl-, Aufzählungs- und Ausschnittstypen, die allesamt in beliebiger Kombination wandelbar sind. So können Sie z. B. „Char(n)" als Ersatz für die Funktion „Chr(n)" oder „Integer(x)" statt „Ord(x)" verwenden.

Das sollten Sie aber besser sein lassen, denn für solche Sachen gibt es ja die erwähnten Standard-PASCAL-Funktionen, während Typkonvertierungen eine Spezialität von MaxonPASCAL sind. Interessant ist allerdings die Möglichkeit, von Integer nach Aufzählungstypen zu konvertieren, also hier „ord" umzukehren:

```
Program Woche;
Type
Tage=(Montag,Dienstag,Mittwoch,Donnerstag,Freitag,Samstag,Sonntag);
Var  i: Integer;  t: Tage;
Begin
  For i:=0 to 6 Do
    Begin t:=Tage(i);
          If t In [Samstag,Sonntag] Then
            writeln('Wochenende!')
    End
End.
```

Desweiteren kann man alle geordneten Datentypen in Pointer und zurück wandeln. Auch in Fließkommatypen lassen sie sich wandeln („Real('A')" würde z. B. die Real-Zahl 65 liefern), aber nicht umgekehrt.

Pointer kann man, wie erwähnt, in andere Pointertypen oder in alle geordneten Datentypen konvertieren. „Boolean(NIL)" wäre z. B. eine besonders umständliche Schreibweise für „false".

Die Datentypen „Real" und „Double" (bzw. „LongReal") lassen sich lediglich untereinander umwandeln, etwa wenn man eine bestimmte Rechengenauigkeit erzwingen will.

Hier ist eine Tabelle mit allen direkten Konvertierungsmöglichkeiten:

		nach:		
		geordnet	Pointer	Fließkomma
	geordnet	X	X	X
von:	Pointer	X	X	
	Fließkomma	•		X

(X = möglich, • = mit Umweg über „Trunc" und „Round")

5.10 Literale für Records und Arrays

Das AMIGA-Betriebssystem wurde in der Sprache C geschrieben. Dies hatte unter anderem zur Folge, daß geradezu exzessiv mit Structs (=Records) und über Zeiger verketteten Listen gearbeitet wurde.

Wenn Sie von PASCAL aus die Betriebssystemfunktionen nutzen wollen, müssen Sie deshalb immer und immer wieder Records initialisieren.

Ein Beispiel: Um mit der Intuition-Funktion „OpenWindow" ein Fenster zu öffnen, benötigt man eine Variable des Typs „NewWindow", der wie folgt definiert ist:

```
TYPE
  NewWindow = RECORD
   LeftEdge,TopEdge,Width,Height: integer;
   DetailPen,BlockPen: Byte;
   IDCMPFlags,Flags: Long;
   FirstGadget,Checkmark: ptr;
   Title: str;
   Screen,BitMap: ptr;
   MinWidth,MinHeight,MaxWidth,MaxHeight,_Type:integer
  END;
```

Jedem der Felder müssen Sie jetzt einen Wert zuweisen. In „normalem" PASCAL sähe das etwa so aus:

```
VAR Neu: NewWindow; ...

Neu.LeftEdge:=10; Neu.TopEdge:=20; Neu.Width:=620; ...
```

...oder so:

```
WITH Neu DO
  BEGIN
    LeftEdge:=10; TopEdge:=20; Width:=620; ...
  END;
```

Das ist natürlich sehr mühsam. Weil dies aber so oft nötig ist, bietet MaxonPASCAL Ihnen hier eine interessante Hilfe: Wertzuweisung auf einen Schlag! In unserem Beispiel sähe das so aus:

```
Neu:=NewWindow(10,20,6 0,160,$200,$100f,Nil,Nil,'Fenster',Nil,Nil,
               200,50,640,200,1);
```

Und dann ist die ganze Struktur schon initialisiert! An diesem Beispiel sehen Sie, wie die Syntax aussieht: Erst kommt der Typbezeichner und dann in runden Klammern, jeweils durch Komma getrennt, die Werte, welche die einzelnen Felder erhalten sollen. Falls Sie sich mit dem Amiga-Betriebssystem noch nicht so gut auskennen und Ihnen die Bedeutung dieses Records deshalb unklar ist: betrachten Sie es „'halt als irgendso 'n Record". Das vorher beschriebene geht natürlich mit jedem Recordtypen, also nicht nur mit denen, die zum Betriebssystem gehören, sondern auch mit den von Ihnen selbstdefinierten.

Welche Vor- und Nachteile hat das Ganze? Nun, es ist schön kurz. Andererseits müssen Sie die jeweilige Struktur genau kennen, denn hier kommt es auch auf die Reihenfolge der Felder an. Das ist aber nicht so schlimm: bei der „normalen" Record-Initialisierung müssen Sie die Struktur ebenfalls genau kennen, um zu vermeiden, daß Sie ein Feld vergessen. Der wohl gravierendste Nachteil ist, daß diese Syntax zu keinem anderen PASCAL kompatibel ist, schon gar nicht zu ISO-Standard-PASCAL. Ihre Programme sind dadurch nicht mehr auf andere Rechner oder Compiler übertragbar. Deshalb sollten Sie (im Rahmen der Aktion „sauberer programmieren") diese Art der Zuweisung nur im Zusammenhang mit AMIGA-Betriebssystem-Funktionen verwenden, die ja ohnehin nicht auf andere Rechnersysteme übertragbar sind. Hier ist es wohl auch am nötigsten.

Bei Records mit einem sog. Variant Part (mit „CASE") ist diese Art der Wertzuweisung nicht möglich. Dafür geht es aber bei Arrays ganz genau so:

```
PROGRAM ArrayDemo;
TYPE   Arr = ARRAY[1..10] of integer;
VAR
  Feld: Arr;
  i: integer;
  BEGIN
    Feld =Arr(17,4,21,47,11,42,0,8,15,26731);
    FOR i:=1 TO 10 DO
    write(Feld[i]:7)
  END.
```

Beachten Sie bitte, daß auch hier runde Klammern zu verwenden sind.

Mit einem „ARRAY ... OF CHAR" geht das übrigens nicht - ist aber auch gar nicht nötig, denn das ist ja ein String, und dafür gibt es bekanntlich viel einfachere Möglichkeiten der Wertzuweisung.

Seit Version 2.0 von MaxonPASCAL existiert auch die Möglichkeit, mehrdimensionalen Arrays und geschachtelten Records auf diese Weise einen Wert zuzuweisen. Dazu verwendet man einfach eine Klammerebene mehr.

Beispiel:

```
PROGRAM Array2Demo;
TYPE Arr2 = Array [ 1..3, 0..1 ] OF integer;
VAR Feld2: Arr2;
BEGIN
  Feld2:= Arr2((17,4), (42,0), (8,15)); );
END.
```

Denken Sie daran, daß dieses zweidimensionale Array lediglich eine Abkürzung für „ARRAY [1..3] OF ARRAY [0..1] OF integer" ist. So wird dann auch verständlich, daß wir hier drei Zahlenpaare haben und nicht etwa zwei Tripel. Bei höheren Dimensionsanzahlen geht das analog.

Analog behandeln wir auch geschachtelte Records:

```
PROGRAM RecRecRecDemo;
TYPE Rec = RECORD
             Name: RECORD
                     Vor,Nach: STRING
                   END;
             Adresse: RECORD
                        Strasse: RECORD
                                   Name: String;
                                   Hausnummer: integer
                                 END;
                        Ort: RECORD
                               Plz: integer;
                               Name: String
                             END
                      END
           END;
VAR Person: Rec;
BEGIN
  Person:= Rec(('Zaphod', 'Beeblebrox'),
               (('Industriestraße', 26), (65760, 'Eschborn')));
  WITH Person DO
    BEGIN
      writeln(Name.Vor, ' ', Name.Nach);
      writeln(Adresse.Strasse.Name, ' ', Adresse.Strasse.Hausnummer);
      writeln(Adresse.Ort.Plz, ' ', Adresse.Ort.Name)
    END
END.
```

Als kleines Extra funktioniert diese Schachtelung auch bei Records, die Arrays enthalten, und bei „Arrays of Records".

Ein Beispiel für letzteres:

```
PROGRAM ArrRecDemo;
TYPE KomplexVektor = ARRAY [1..5] OF RECORD
                                       Re, Im: Real
                                     END;
```

```
VAR x: KomplexVektor;
BEGIN
  x:= KomplexVektor((0,1), (-3,2), (42,0), (8,15), (17,4));
END.
```

5.11 Dateitypen

Eine Datei wird in PASCAL durch eine Variable repräsentiert, und die braucht auch einen Datentypen: einen FILE-Typen.

Die Syntax ist denkbar einfach:

```
Dateityp = "FILE" [ "OF" Typ ]
```

„Typ" gibt dabei den Typ der Daten an, die in der Datei gespeichert werden sollen. Dabei darf es sich nicht um FILE-Typen handeln - was sollte denn wohl auch „FILE OF FILE OF..." darstellen?

Es gibt einen vordefinierten FILE-Typ: „Text" ist „FILE OF Char". Und damit wären wir auch schon bei den beiden grundlegend verschiedenen Dateitypen: Es gibt Textdateien und Nicht-Textdateien. In Textdateien können Daten vieler unterschiedlicher Typen ein- und ausgegeben werden: Char, String, Boolean (nur Ausgabe) und alle Zahlentypen, und zwar bei Bedarf alle in dieselbe Datei. Bei der Ein- und Ausgabe werden die Daten als Folge von ASCII-Zeichen repräsentiert.

Nicht-Textdateien, also solche, bei denen der Elementtyp nicht „Char" ist, sind an ihren festen Elementtypen gebunden. Dafür ist dieser aber beliebig (außer, wie bereits erwähnt, Dateitypen). In solche Dateien können also nur Daten eines ganz bestimmten Typs ausgegeben bzw. aus ihnen eingelesen werden. Die Daten werden in solchen Files im jeweiligen nicht unmittelbar lesbaren internen Datenformat abgespeichert.

Grob gesehen ist eine Datei eine beliebig lange Folge von Daten. Ein Schreib- oder Lesezeiger wandert von vorn nach hinten durch diese Datei. Auf das Dateielement, auf dem der Zeiger gerade steht, kann man durch die Puffervariable zugreifen: Ist eine Datei als

```
VAR Dat: FILE OF Typ;
```

deklariert, so heißt die Puffervariable dazu „Dat ^ " und ist vom Datentyp „Typ".

Zuerst muß man aber die Datei fürs Lesen oder Schreiben öffnen:

Reset(Dat)	öffnet die Datei, die bereits existieren muß, zum Lesen.
Rewrite(Dat)	öffnet sie für Schreibzugriffe. Falls sie noch nicht existiert, wird sie erzeugt.

Woher „weiß" das Programm aber, welche reale Datei, also welches File auf der Diskette o. ä., gemeint ist?

Dazu gibt es zwei Möglichkeiten:

- Man kann der Datei VOR dem Aufruf von Reset/Rewrite mit der Prozedur „Assign" einen String als Namen zuweisen. Beispiel:

```
Assign(Dat, 'PASCAL:Textdateien/Text1');
Reset(Dat);
```

  Der Name wird dann nach dem „Assign"-Befehl in einem Puffer aufbewahrt. Achtung: Dazu stehen nur 30 Bytes zur Verfügung, so daß die Dateinamen nicht länger werden können!

- Man kann auch direkt beim Reset/Rewrite einen Dateinamen als zweiten Parameter angeben, z. B. so:

```
Reset(Dat, 'PASCAL:Textdateien/Text1');
```

  Hier darf der Dateiname beliebig lang sein.

Falls die Datei nicht geöffnet werden konnte, hat die Funktion *„IOResult"* einen von 0 verschiedenen Wert, die Fehlernummer. Was nun geschieht, hängt davon ab, ob Sie *„Reset"* oder *„Rewrite"* verwendet haben: Im ersten Fall wurde die Datei zum Lesen geöffnet und enthält in der Regel Daten (es gibt natürlich auch leere Dateien), die Sie jetzt der Reihe nach lesen können, bis die Funktion *„Eof(Dat)"* wahr wird und so das Dateiende anzeigt. Dazu dient die Prozedur *„Get"*:

```
Get(Dat)
```

liest das nächste Element aus der Datei und legt es in der Variablen „Dat^" ab.

```
Get(Dat); Variable:= Dat^;
```

kann ersetzt werden durch:

```
Read(Dat, Variable);
```

Wenn Sie eine Datei mit *„Rewrite"* zum Schreiben geöffnet haben, ist sie stets leer. Sie können nun Daten in die Datei schreiben, indem Sie sie zunächst in der Puffervariablen ablegen und dann mit der Prozedur *„Put"* schreiben.

Beispiel:

```
Dat^:= Ausdruck; Put(Dat);
```

Auch dafür gibt es eine Abkürzung:

```
Write(Dat, Ausdruck);
```

Soviel zur „normalen" Dateihandhabung von Standard-PASCAL. MaxonPASCAL erlaubt aber auch nicht-sequentiellen Dateizugriff, indem man mit der Prozedur *„Seek"* den Zeiger an eine beliebige Position setzen kann. Ferner kann man auf mit *„Reset"* geöffnete Dateien auch schreibend zugreifen, wobei jeweils die Daten, auf denen der Dateizeiger gerade steht, überschrieben werden.

Außerdem gibt es noch typfreie Dateien. Sie werden einfach als „FILE“ ohne Typangabe deklariert, z. B. so:

```
VAR Datei: FILE;
```

In diesem Fall gibt es dann keine Puffervariable („Datei ^ “), so daß auch die Prozeduren *„Get“* und *„Put“* nicht benutzt werden können. Nur mit den Prozeduren *„BlockRead“* und *„BlockWrite“* kann aus typfreien Dateien gelesen bzw. in sie geschrieben werden.

6. Variablen und Konstanten

6.1 Standard-Variablen

MaxonPASCAL stellt folgende Standard-Variablen zur Verfügung:

Input, Output (Text)

Die Standard-Ein- und Ausgabedateien

Zeiger auf die Basisadresse der Dos-Library.

MathBase (Ptr)

Zeiger auf die "Mathffp.library"

MathtransBase (Ptr)

Dieser Pointer zeigt auf die Basisadresse der Mathtrans-Library. Aber Achtung: Diese Library liegt nicht im ROM, sondern muß von der Systemdiskette nachgeladen werden. Deshalb wird sie nur bei Bedarf geöffnet, d. h. sobald eine Funktion daraus aufgerufen wird.

FromWB (Boolean)

Jedes AMIGA-Programm kann im Prinzip auf zwei sehr unterschiedliche Arten gestartet werden: Von der Workbench oder vom CLI. Die Unterschiede liegen zum einen in der Art der Parameterübergabe (beim CLI-Start kann eine Zeichenfolge übergeben werden, beim Benchstart können evtl. andere Icons aktiviert sein) und zum anderen bei der Ein-/Ausgabe (im ersten Fall steht Ihnen dazu das CLI-Fenster zur Verfügung, im anderen Fall müssen Sie sich zuerst ein Fenster öffnen - bitte beachten Sie dazu auch das Demoprogramm "WB.p").

Die Variable "FromWB" gibt Ihnen nun die Möglichkeit, festzustellen, wie Ihr Programm gestartet wurde. Sie ist (klar) "true" genau dann, wenn der Aufruf von der Workbench erfolgte.

Die MaxonPASCAL-Entwicklungsumgebung simuliert übrigens einen CLI- Start, d. h. wenn Sie Ihr Programm von dort starten, hat FromWB stets den Wert "false".

ParameterStr (Str)

Dieser Stringzeiger zeigt auf die Zeichenfolge, die, wie oben erwähnt, beim CLI-Start dem Programm übergeben wurde.

ParameterLen (Integer)

Der Parameterstring endet nicht notwendig mit einem Nullbyte, deshalb enthält diese Variable die Länge.

StartupMessage (Ptr)

Beim Programmstart von der Workbench wird eine Message übergeben, die verschiedene Informationen enthält. Das Laufzeitsystem von MaxonPASCAL holt diese Nachricht ab und beant-

wortet (ReplyMsg) sie am Ende auch, so daß Sie sich darum nicht zu kümmern brauchen. Sie können sie aber mit Hilfe der Pointervariablen "StartupMessage" auswerten.

Mem (Array [0..MaxLongInt] of Byte absolute 0)

Dieses Feld repräsentiert den byteweise organisierten Hauptspeicher des Rechners. Dadurch sind einfache Speicherzugriffe möglich.

6.2 Vordefinierte Konstanten

Es gibt folgende vordefinierte Konstanten:

True	logischer Wert "wahr"
False	logischer Wert "falsch"
MaxInt	größte Integer-Zahl (32767)
MaxLong	größte LongInt-Zahl (2147483647)
MaxLongInt	anderer Name für "MaxLong"
MaxCard	höchste Cardinal-Zahl (65535)
MaxWord	anderer Name für "MaxCard"
MaxShortCard	größte 8-Bit-Cardinalzahl (255)
MaxByte	anderer Name für "MaxShortCard"
MaxShort	größte 8-Bit-Integerzahl (127)
MaxShortInt	anderer Name von "MaxShort"
Pi	= 3.14159265358979

7. Prozeduren und Funktionen

MaxonPASCAL beherrscht alle Prozeduren und Funktionen des Standards - außer „Pack" und „Unpack", denn Datenstrukturen werden automatisch gepackt verwaltet. Daneben gibt es aber noch viele zusätzliche Befehle für alle Bereiche der PASCAL-Programmierung.

7.1 Ein- und Ausgabe

Write [(Write-Parameterliste)]

Syntax für Write-Parameterliste:

```
(Dateivariable | Write-Parameter) { "," Write-Parameter }
```

Syntax für Write-Parameter:

```
Ausdruck [ ":" Ausdruck [ ":" Ausdruck ] ]
```

Dies ist die Standard-Ausgabeprozedur. Sie kann prinzipiell ohne Parameterliste verwendet werden - dann gibt sie nichts aus und ist also überflüssig. Die Parameterliste ist eine geklammerte Liste von durch Komma getrennten „Write-Parametern" (siehe Syntax), von denen der erste auch eine Variable eines Filetyps sein kann. Falls letzteres nicht der Fall ist, so ist „Output" Default-Ausgabedatei.

Ist die Ausgabedatei vom Typ „Text", also „FILE OF Char", so haben Sie viele verschiedene Ausgabemöglichkeiten: die nachfolgenden Parameter dürfen von numerischen, booleschen, Char- und Stringtypen sein. In jedem Fall darf hinter dem Write-Parameter, mit einem „:" getrennt, die gewünschte Feldbreite für rechtsbündige Ausgabe stehen.

Fehlt sie, so gibt MaxonPASCAL grundsätzlich linksbündig aus. Bei der Ausgabe von Real-Werten darf man auch noch einen zweiten Parameter für das gewünschte Ausgabeformat angeben. Näheres dazu finden Sie im Abschnitt über Gleitpunkttypen.

Wenn die spezifizierte Ausgabedatei keine Textdatei ist, so müssen alle nachfolgenden Parameter zum Elementtypen der Datei typkompatibel sein. In eine Datei des Typs „FILE OF Real" könnte man z. B. einen beliebigen numerischen Wert schreiben. Wenn „f" eine Dateivariable eines solchen Typs ist, so wäre

Write (f, x, y)

äquivalent zu:

```
f^:= x ;  Put (f);
f^:= y ,  Put (f);
```

Näheres zu „*Put*" finden Sie im Kapitel 7.2 „Dateien".

WriteLn [(Write-Parameterliste)]

Write-Parameterliste: Wie bei „*Write*"

Diese Prozedur ähnelt „*Write*", ist aber nur bei Textdateien möglich. Nach Ende der Ausgabe wird eine neue Zeile angefangen. Hier ist es desöfteren sinnvoll, die Parameterliste ganz wegzulassen: Dadurch wird ein Zeilenvorschub in der Standard-Ausgabe bewirkt.

Read [(Read-Parameterliste)]

Syntax für Read-Parameterliste:

```
Variable { "," Variable }
```

Die Standard-Eingabeprozedur. Wie bei „*Write*" kann auch hier die Parameterliste weggelassen werden, was aber keinen sinnvollen Befehl ergibt. Die Parameterliste, sofern vorhanden, ist eine Liste von durch Kommata getrennten Variablen. Analog zu „*Write*" darf auch hier der erste Parameter eine File-Variable sein, sonst wird „*Input*" als Default-Eingabekanal benutzt. Auch hier gibt es eine Unterscheidung zwischen Text- und Nicht-Text-Dateien:

Ist die Eingabedatei vom Typ „Text" (wie z. B. die Datei „Input"), so sind als Parameter Variablen von numerischen, Char- und Stringtypen erlaubt. Es werden dann jeweils Zeichen aus der Eingabedatei gelesen und in den gewünschten Typen konvertiert.

Bei Nicht-Text-Eingabedateien müssen alle Parameter Variablen sein, die zum Elementtyp der Datei typkompatibel sind, insbesondere dieselbe „Breite" haben müssen. Hier werden direkt die Daten und nicht eine lesbare Repräsentation der Daten gelesen.

Die Dateihandhabung von Standardpascal ist nicht mehr ganz zeitgemäß und mußte bei MaxonPASCAL leicht variiert werden:

```
Read(datei,variable)
```

entspricht deshalb:

```
Get(datei) ; Variable:= datei^
```

und nicht (Wirth: PASCAL Report, Kap. 11.4.1):

```
Variable:= datei^ ; Get(datei)
```

Desweiteren wird „*EoF*" erst „true", wenn das Dateiende wirklich erreicht wurde, d. h. eventuell (bei Textdateien) erst dann, wenn nichts mehr gelesen werden konnte und die Variable dadurch undefiniert ist. Dies läßt sich in der Regel durch Verwendung von „*SeekEoF*" statt „*EoF*" vermeiden.

ReadLn [(Read-Parameterliste)]

Wie Read, aber nur bei Textdateien möglich.

```
ReadLn(parameter)
```

entspricht

```
Read(Parameter); ReadLn
```

und „*ReadLn*" entspricht wiederum

```
While not eoln(datei) do Get(datei)
```

Im Klartext: Alles, was in der Eingabezeile noch hinter dem gelesenen Teil steht, wird überlesen.

Page [(datei)]

datei: Variable eines File-typs, Default ist „Output"

Erzeugt Seitenvorschub oder -wechsel. Beim Bildschirm bedeutet dies Löschen.

ClrScr [(datei)]

datei: Text-File

Anderer Name von „*Page*" (wurde aus Gründen der Kompatibilität implementiert).

GotoXY (x,y)

x, y: Integer

Setzt den Cursor in Spalte x, Zeile y. Die linke obere Ecke des Ausgabefensters hat die Koordinaten (1,1).

ClrEol

löscht die Zeile ab Cursor.

InsLine

fügt auf dem Bildschirm eine Zeile ein, d. h. der Text unterhalb der Zeile, in der der Cursor steht, wird nach unten gescrollt.

DelLine

Gegenteil von „*InsLine*": Die Zeile, in der der Cursor steht, wird gelöscht, alles, was darunter steht, um eine Zeile nach oben gescrollt.

EoF [(datei)]: Boolean

datei: Variable eines Filetyps, Default ist „input"

Dateiende- und Dateifehler-Funktion. EoF(input) ist übrigens immer „false".

Eoln [(datei)]: Boolean

datei: Variable des Typs „text", Default ist „input"

Meldet Zeilenende bei Read/ReadLn aus Textdateien.

SeekEoLN [(datei)]: Boolean

datei: Textfile-Variable, Default ist „input".

Diese Funktion entspricht dem normalen „EoLN", allerdings werden hier Leerzeichen überlesen. Beispiel:

```
Var i, j: integer;
Begin
  write('Bitte Zahlen eingeben: '); i:=0;
  Repeat
   read(j); i:=i+j
  Until EoLN;
  writeln('Summe: ',i)
End.
```

Dieses Programm liest von der Standardeingabe eine Zeile voller Zahlen und berechnet die Summe der Zahlen. Problem: Wenn der Benutzer hinter der letzten Zahl noch ein Leerzeichen eingibt, wird das Zeilenende nicht erkannt. Das kann behoben werden, indem man „EoLN" einfach durch „SeekEoLN" ersetzt.

SeekEoF [(datei)]: Boolean

datei: Textfile-Variable, Default ist „input".

Analog zu „SeekEoLN" prüft diese Funktion „EoF", wobei Leerzeichen und Zeilenenden überlesen werden. Im Gegensatz zum normalen EOF arbeitet „SeekEoF" nur auf Textfiles.

EmptyLN [(datei)]: Boolean

datei: Textfile-Variable, Default ist „input".

Diese Funktion liest eine Zeile aus der Textdatei in den Puffer, überliest führende Leerzeichen und stellt dann fest, ob die Zeile leer ist.

Anwendungsbeispiel: „*Read*" wartet beim Lesen von Zahlen stur darauf, daß eine Zahl eingegeben wird - da hilf kein RETURN-Drücken. Mit der EmptyLN-Funktion können Sie zuvor prüfen, ob der Benutzer etwas einzugeben gedenkt.

Beispiel:

```
Var i: integer;
Begin
  i:=$0815;
  writeln('Alter Wert: ',i);
  write('Neuen Wert eingeben oder mit RETURN bestätigen: ');
  If not EmptyLN Then readln(i);
  writeln('Neuer Wert ist: ',i)
End.
```

In dem Moment, wo die Funktion „*EmptyLN*" aufgerufen wird, wartet das Programm darauf, daß der Benutzer etwas eingibt und RETURN drückt, und es wird geprüft, ob die Eingabe leer ist, d. h. ob er einfach nur „RETURN" getippt hat. Die Funktion „*EmptyLN*" liest selbst aber noch nichts - abgesehen von Leerzeichen, die überlesen werden.

Nur wenn in der eingegebenen Zeile wirklich etwas steht, liest das obige Programm dies mit „Readln(i)" ein, andernfalls behält „i" den alten Wert.

7.2 Dateien

Reset (datei [, name])

datei: Variable eines FILE-Typs
name: Stringkonstante oder -Variable

Eine Datei wird zum Lesen und Schreiben geöffnet. Dabei kann - als Erweiterung des ISO-Standards - optional ein Dateiname angegeben werden. Wird er weggelassen, so muß zuvor mit dem „*ASSIGN*"-Befehl ein Dateiname zugewiesen worden sein.

Reset in der erweiterten Form (d. h. mit Dateinamen) übernimmt den String nicht in den Puffer, so daß einerseits keine Längenbeschränkung besteht, andererseits aber keine Stringausdrücke verwendet werden können.

Die Datei muß bereits existieren. Konnte sie nicht geöffnet werden, so wird kein Fehler gemeldet! Vielmehr gilt dann „Eof(datei)=true" und „IOResult<>0". Sie müssen den Fehler also selbst abfangen, bzw. das Laufzeitsystem meldet bei der nächsten Ein- oder Ausgabeoperation auf dieser Datei einen „File not open"-Error.

Rewrite (datei [, name])

datei: Variable eines FILE-typs
name: Stringkonstante oder -Variable

Eine Datei wird für Schreibzugriff geöffnet. Nach erfolgreichem *„Rewrite"* ist die Datei völlig leer, und der Lesezeiger steht am Anfang. Dies ist der (wichtige) Unterschied zu *„Reset"*, wonach man außer lesen zwar auch schreiben kann, aber dabei lediglich den alten Dateiinhalt überschreibt.

Get (datei)

datei: Variable eines Dateityps

Daten in Puffer „datei ^ " lesen.

Put (datei)

datei: Variable eines File-Typs

Puffer „datei ^ " in Datei schreiben.

Close (datei)

datei: Variable eines File-Typs

Wenn Sie eine Datei mit *„Reset"* oder *„Rewrite"* geöffnet haben, müssen Sie sie nach Gebrauch mit dem „Close"-Befehl wieder schließen. Falls Sie das nicht tun, wird es am Programmende automatisch durchgeführt.

Assign (datei, name)

datei: Variable eines FILE-typs
name: Stringausdruck

Einer Datei wird ein externer Dateiname zugewiesen. Dies ist nötig, damit das Laufzeitsystem beim anschließenden *„Reset"* oder *„Rewrite"* dieser Datei weiß, welche reale Datei angesprochen werden soll.

Der Dateiname wird in einen zur Filevariablen gehörenden Puffer übernommen, der 30 Bytes lang ist. Mit dem Nullbyte am Ende kann ein mittels *„Assign"* zugewiesener Dateiname also maximal 29 Zeichen lang sein. Falls das nicht reicht, gibt es alternativ die erweiterten Versionen von *„Reset"* und *„Rewrite"* (s. u.), die ohne vorheriges *„Assign"* auskommen.

IOResult: integer

Nach jeder Ein- und Ausgabeoperation von MaxonPASCAL, also z. B. *„Reset"*, *„Rewrite"*, *„Read"*, *„Seek"* usw. wird diese Funktion mit der vom Betriebsystem gelieferten Fehlernummer initialisiert, also „0", wenn alles klar ist, und einen Wert „>0", wenn ein Fehler auftrat.

Beispiel:

```
Var f: text;
Begin reset(f,'yxlegrümpf'); writeln(IOResult) End.
```

liefert, falls nicht zufällig eine Datei namens „yxlegrümpf" existiert, die Fehlernummer 205: „Object not found".

Filehandle (datei): LongInt

datei: Dateivariable

Diese Funktion liefert zu einer PASCAL-Dateivariablen die zugehörige Dateihandle, so daß Sie auf die Datei auch mit den Befehlen der Dos.library arbeiten können. Beispiel:

```
{$path "pascal:include/"; incl"libraries/dos.h" }

Var f: text;
    e: integer;
    c: Char;
Begin
  reset(f, 'CON:0/0/640/200/Kein CLI');  { Fenster öffnen }
  If IOResult=0 Then exit;               { Fehler abfangen! }
  e:= Execute('dir'$a'endcli',         { Befehlsfolge }
              filehandle(f),           { Eingabedatei }
              filehandle(f));          { Ausgabedatei }
  writeln(f);
  writeln(f,'Fehlercode war:',e);
  writeln(f,'RETURN drücken!');
  readln(f,c);
  close(f)
End.
```

Dieses Programm öffnet mit *„Reset"* ein Fenster und läßt darin vom DOS die Befehle *„Dir"* und *„EndCLI"* ausführen (letzterer ist wichtig, sonst hängt das DOS sich auf!). Die Befehle müssen durch das Zeichen #$a, also ein *„Linefeed"*, getrennt werden.

Beachten Sie bitte: - *„Filehandle"* prüft nicht, ob die Datei auch wirklich geöffnet wurde. Falls dies nicht der Fall sein sollte, ist das Ergebnis undefiniert.

☞ Wenn das Programm aus dem PASCAL-System oder von der Workbench gestartet wird, sind „Filehandle(input)" und „Filehandle(output)" undefiniert. Wenn der Programmstart vom CLI erfolgte, liefern die beiden Ausdrücke die Handle der Standardein- bzw. Ausgabedatei.

☞ Mit DOS-Aufrufen können Sie die interne Dateihandhabung des PASCAL durcheinanderbringen. Generell sind „*EoF*", „*EoLN*", „*FilePos*" und „*FileSize*" einer Datei nach einem derartigen Aufruf undefiniert.

„DosClose(Filehandle(...))" ist absolut tabu - Gurugefahr!

Seek (datei, position)

datei: File-Variable
position: Ganzzahliger Ausdruck

Setzt den Schreib-/Lesezeiger einer zuvor mit „*Reset*" geöffneten Datei auf eine bestimmte Position. Die erste Position hat die Nummer 0.

Eine „Position" ist ein Element des Datei-Grundtyps, bei „Text" also ein Char (=1 Byte), bei „File of integer" ein 2-Byte-Wort usw.

Nach „Seek(f)" ist die Puffervariable „f^" noch nicht initialisiert. Es muß also (wenn's unbedingt sein muß) noch „Get(f)" aufgerufen werden.

Filepos (datei): LongInt

datei: File-Variable

Gibt die Position des Schreib-/Lesezeigers in einer zuvor mit „*Reset*" geöffneten Datei an. Nach „Seek(datei,n)" liefert „Filepos(datei)" beispielsweise den Wert n, vorausgesetzt, die Datei ist entsprechend lang.

Filesize (datei): LongInt

datei: File-Variable

Liefert die Größe einer zuvor mit „*Reset*" oder „*Rewrite*" geöffneten Datei. Im allgemeinen wird die Größe aber nicht in Bytes oder Blöcken angegeben, sondern in Elementen des Datei-Grundtyps, so daß diese Funktion ohne Umformung zu „*Seek*" und „*FilePos*" kompatibel ist. Beispiel:

```
Var f: Text;
Begin
  reset (f, 's:startup-sequence');
  seek (f, filesize(f));
  writeln (f, 'echo "Hallo!"');
  close (f)
End.
```

Dieses Programm öffnet die Startup-sequence, setzt den Zeiger auf das Dateiende und hängt dort mittels „*writeln*" einen CLI-Befehl an.

Buffer (datei, groesse)

datei : Variable eines beliebigen FILE-Typs
groesse: gewünschte Puffergröße in Bytes

Die Dateizugriffe eines Programms erfolgen intern über die DOS-Library des Betriebssystems, die das Filesystem benutzt, welches wiederum über Devices (Gerätetreiber) auf die Hardware zugreift. Wir haben es also mit einem ziemlichen Aufwand zu tun, der jedesmal getrieben wird, bevor auch nur ein einziges Byte gelesen oder geschrieben werden kann.

Schlimmer noch: Dieser Überbau muß bei jeder einzelnen Dateioperation erledigt werden und ist vom Umfang der anschließend zu übertragenen Daten unabhängig. Sie können sich denken (oder mußten es bereits erfahren), was dann bei Textdateien, bei denen ja zeichenweise gelesen und geschrieben werden muß, passiert: Für jedes übertragene Byte muß die gesamte Maschinerie in Gang gesetzt werden. Selbst die wohl kaum als überragend schnell zu bezeichnenden AMIGA-Diskettenlaufwerke sind dann so schnell, daß die eigentlichen Diskettenzugriffe verglichen mit der enormen Zeitverschwendung durch das Betriebssystem kaum noch Zeit brauchen. Vielleicht kennen Sie noch das IFF-Ladeprogramm, das bei KICKPASCAL 1.0 als Beispielprogramm beilag: Es war scheußlich langsam, aber nicht etwa, weil KICKPASCAL so schlechten Code erzeugt hätte, sondern weil die Bilddaten im wesentlichen byteweise gelesen werden mußten.

Viel effektiver wäre es doch, für eine Datei einen großen Puffer im RAM einzurichten. Bei Schreiboperationen könnte man die Daten erst einmal in diesem Puffer ablegen und dann in einem Rutsch (und mit einem einzigen DOS-Aufruf) schreiben, sobald der Puffer voll ist.

Umgekehrt könnte man bei Leseoperationen zuerst einen großen Datenblock in den Puffer lesen und dann von dort ohne weitere DOS-Benutzung die Daten weiterverarbeiten.

Jaaa, werden Sie jetzt vielleicht sagen, das klingt zwar alles recht sinnvoll und einleuchtend, aber die Verwaltung eines solchen Pufferspeichers bedeutet doch einen hohen Programmieraufwand...? Keine Panik - MaxonPASCAL kann mit der Prozedur *„Buffer"* solche Speicher vollautomatisch (und für Ihr übriges Programm unsichtbar) verwalten. Die Prozedur benötigt nur zwei Parameter: zum einen eine Dateivariable, die zuvor mit *„Reset"* oder *„Rewrite"* geöffnet worden sein muß, und zum anderen die gewünschte Puffergröße in Bytes. Hier bringen wenige kBytes schon eine deutliche Geschwindigkeitserhöhung; über 10000 Bytes zu gehen, ist in der Regel nicht sinnvoll. Das Programm läuft dann ganz genau wie zuvor, nur daß eben die Dateioperationen beschleunigt werden.

Dazu noch ein paar Hinweise:

- Wurde die Datei mit *„Reset"* geöffnet, wird der Puffer nur für Leseoperationen benutzt und bei jedem *„Write"* geleert.
- Nach *„Seek"* wird der Lesepuffer automatisch entleert. Deshalb kann es passieren, daß ein Programm sogar verlangsamt wird, wenn es in einer Datei dauernd hin und her springt.

☞ Zwischen Öffnen und Schließen einer Datei kann die Puffergröße für diese Datei höchstens einmal festgelegt werden. Weitere Aufrufe von „*Buffer*" werden ignoriert.

BlockRead (datei, variable, anzahl)

datei: zum Lesen geöffnete Dateivariable
variable: beliebige Variable, in die Daten gelesen werden sollen
anzahl: Anzahl der zu lesenden Datenblöcke

Oft ist der Inhalt einer Datei nicht so strukturiert, daß man sie sinnvoll als „FILE OF <Typ>" deklarieren könnte. Man denke nur an die diversen IFF-Formate, die aus unterschiedlich aufgebauten Hunks verschiedenster Länge bestehen. Deshalb gibt es die Prozedur „*BlockRead*", die aus einer beliebigen Datei Daten beliebiger Art und Länge in eine beliebige Variable liest. „anzahl" ist dabei die Anzahl der Datenblöcke, die jeweils dem Elementtyp der Datei entsprechen. Ist sie als „FILE OF Typ" deklariert, so werden anzahl * SizeOf(Typ) Bytes gelesen. Bei einer Textdatei entspräche ein Datenblock also einem einzelnen Zeichen, bei typfreien Dateien wird als Datensatztyp „Byte" angenommen.

Beispiel:

```
VAR t: Text;
    s: String[1000];
BEGIN
  Reset (t, 's:Startup-Sequence');
  If FileSize (t) >= 1000 Then
   Error ('Datei ist zu lang!');
  BlockRead (t, s, FileSize (t));
  s [FileSize (t) + 1]:= chr(0);    { Stringende markieren }
  Writeln(s)
END.
```

Das Programm liest die komplette Startup-Sequence in eine Stringvariable und gibt sie aus.

BlockWrite (datei, variable, anzahl)

datei: zum Lesen geöffnete Dateivariable
variable: beliebige Variable, in die Daten gelesen werden sollen
anzahl: Anzahl der zu lesenden Datenblöcke

Analog zu „*BlockRead*" schreibt „*BlockWrite*" Daten aus einer Variablen beliebigen Typs in eine Datei.

7.3 Arithmetische und andere Funktionen

Alle in diesem Abschnitt vorgestellten Funktionen sind Bestandteil des Jensen-Wirth-Standards. Einige von ihnen mußten natürlich für die zusätzlichen Datentypen von MaxonPASCAL erweitert werden. Sie sind in alphabetischer Reihenfolge aufgeführt.

Abs(num): Numerisch

num: Real- oder Ganzzahlausdruck

Betrag von Gleitpunkt- oder Ganzzahldaten. Das Ergebnis ist stets vom selben Typ wie der Parameter.

ArcTan(x): Real

x: Real-Ausdruck

Arcustangens-Funktion. Wie die meisten Real-Funktionen, wird auch diese der Mathtrans.library entnommen, welche bei Bedarf geöffnet wird. Die Library liegt aber nicht im ROM, sondern muß vom „libs:"- Verzeichnis der Systemdiskette geladen werden. Ist diese nicht vorhanden, wird ein Fehler gemeldet.

Chr(i): Char

i: Integer-Ausdruck, Bereich 0..255

Liefert das Zeichen mit dem Ascii-Code i. Umkehrfunktion ist „ord(c)".

Cos(x): Real

x: Real-Ausdruck

Die wohlbekannte Cosinus-Funktion. Bei dieser und allen anderen trigonometrischen Funktionen müssen die Winkel - wie in der Mathematik üblich - im Bogenmaß angegeben werden. Aus einem Gradwinkel erhält man die zugehörige Bogenmaßzahl, indem man durch 180 teilt und mit Pi multipliziert - die andere Richtung geht entsprechend genau andersrum.

Exp(x): Real

x: Real-Ausdruck

Dies ist die Exponentialfunktion zur Basis e (Euler'sche Zahl). Die zugehörige Umkehrfunktion ist *„Ln"*.

Ln(x): Real

x: Real-Ausdruck, x>0

Ergibt den natürlichen Logarithmus von x.

Odd(i): Boolean

i: Ganzzahl-Ausdruck

Odd(i) ist „true" genau dann, wenn i ungerade ist.

Ord(o): Integer oder anderer Ganzzahltyp

o: Ausdruck eines geordneten Typs (Ganzzahl, Boolean, Char, Aufzählungstyp)

Liefert die Ordnungszahl des Ausdrucks in den Klammern. Ist dieser Ausdruck von einem Ganzzahltyp, ist das Ergebnis vom selben Typ und auch vom selben Wert (so daß die Funktion hier unsinnig ist). Andernfalls ist das Ergebnis vom Typ „Integer".

Pred(o): ordinal

o: Wie bei „Ord(o)"

Die Vorgänger-Funktion. Das Ergebnis ist stets vom selben Typ wie der Parameter und ist der Wert, dessen Ordnungszahl um 1 kleiner ist als der des Parameters. Beispiel:

```
Pred(26731)=26730, Pred('Y')='X', Pred(true)=false.
```

Pred(false) oder Pred(chr(0)) sind eigentlich undefiniert, es kommt aber zu keiner Fehlermeldung.

Round(x): LongInt

x: Real-Ausdruck, abs(x)<=MaxLongInt

Gerundeter Wert von x.

Sin(x): Real

x: Real-Ausdruck

Die beliebte Sinus-Funktion. Natürlich betrachtet auch sie den Parameter als Winkel im Bogenmaß (vgl. „Cos(x)").

Sqr(num): numerisch

num: Numerischer Ausdruck

Berechnet das Quadrat des Parameters. Das Ergebnis ist stets vom selben Typ wie der Parameter.

Sqrt(x): Real

x: Real-Ausdruck, x>=0

Berechnet die Quadratwurzel des Parameters

Succ(o): ordinal

o: wie bei „ord(o)" und „Pred(o)"

Diese Funktion ist das Gegenstück zu „*Pred*", d.h. sie liefert den Nachfolger eines Ausdrucks eines geordeneten Datentyps.

Trunc(x): LongInt

x: Real-Ausdruck, abs(x) <= MaxLongInt

Liefert den Ganzzahl-Anteil von „x". Im Gegensatz zu „*Round*" wird hier nicht gerundet; vielmehr werden die Nachkommastellen einfach weggelassen.

7.4 Pointer

New(point)

point: Variable eines Pointertyps

Speicherplatz für eine dynamische Variable reservieren und „point" darauf setzen.

Bei Records mit Variantenteil wird immer die maximale Größe reserviert. Die Version „New(point,feld1,feld2,...)" ist nicht implementiert.

Dispose(point)

point: Variable eines Pointertyps

Dynamische Variable löschen, Speicher freigeben.

DisposeAll

Alle dynamischen Variablen und alle von *AllocMem* reservierten Speicherbereiche werden freigegeben. Dies ist keine Standard-PASCAL- Prozedur.

7.5 AMIGA-Spezifisches

Alloc_Mem (len, cond): LongInt

len (LongInt): gewünschte Größe in Bytes
cond (LongInt): Bedingungen

Diese Funktion entspricht im Prinzip der Exec-Funktion *„AllocMem"*.

Der AMIGA ist - wie Ihnen wohl kaum entgangen sein dürfte - multitaskingfähig, d.h. mehrere Programme können gleichzeitig laufen.

Das führt natürlich dazu, daß ein Programm nicht so ohne weiteres auf den Speicher zugreifen kann, denn es besteht ja immer die Gefahr, daß sich dabei zwei Programme ins Gehege kommen. Also muß sich ein Programm bei Bedarf vom Betriebssystem Speicher reservieren lassen. Genau das tut nun AllocMem bzw. Alloc_Mem. Dabei ist „Len" die gewünschte Größe des Speichers.

Bei „Cond" können Sie bestimmte Bedingungen angeben. Eine 2 (genannt MEMF_CHIP) bewirkt z. B., daß der Speicher im sog. Chipmemory liegt, bei $10000 (MEMF_CLEAR) wird der Speicher automatisch gelöscht (mit Nullen gefüllt). Wenn Sie keine besonderen Ansprüche stellen, geben Sie einfach cond=0 an.

Das Ergebnis der Funktion ist die Anfangsadresse des soeben reservierten Speichers. Was unterscheidet nun diese MaxonPASCAL-Funktion von der fast gleichlautenden Exec-Betriebssystem-Funktion?

Zum einen steigt Alloc_Mem mit einer Fehlermeldung aus, wenn kein Speicher reserviert werden konnte (z.B. weil schon alles belegt war). Zum anderen führt das Laufzeitsystem von MaxonPASCAL über die Alloc_Mem-Funktion quasi Buch, so daß alle reservierten Speicherbereiche am Programmende automatisch wieder freigegeben werden. Das können Sie aber auch „manuell" machen: Der entsprechende Befehl heißt Free_Mem.

Open_Window (X, Y, W, H, Farbe, IDCMP, Flags, Name, Screen, MinW, MinH, MaxW, MaxH): Ptr

X (Integer): x-Koordinate der Fensterposition
Y (Integer): y-Koordinate der Position
W (Cardinal): Breite („Width")
H (Cardinal): Höhe („Height")
Farbe (Cardinal): Lo-Byte ist Hinter-, Hi-Byte Vordergrundfarbe
IDCMP (Long): Signal-Bit-Maske
Flags (Long): z.B. $1000 für ACTIVATE, 8 für WINDOWCLOSE
Name (Str): Fenstertitel
Screen (Ptr): Zeiger auf Screen, „Nil" für Workbenchscreen
MinW, MinH (Cardinal): Mindestbreite und -höhe

MaxW, MaxH (Cardinal): Maximalbreite und -höhe

Erraten: Open_Window öffnet ein Fenster. Wieder entsprechen die einzelnen Parameter den Feldern der NewWindow-Struktur, wie zuvor bricht das Laufzeitsystem ab, falls das Fenster nicht geöffnet werden konnte, und abermals werden die Fenster am Programmende automatisch geschlossen.

„FirstGadget", „Checkmark" und „BitMap" werden jeweils auf Nil gesetzt, „type" je nach Sachlage auf 1 oder 15.

Close_Window (p)

p: Ptr

Dies ist das Gegenteil von *„Open_Window"*: ein Fenster wird geschlossen.

Free_Mem (Adr,Len)

Adr (LongInt): Speicheradresse
Len (LongInt): Länge in Bytes

Dies ist das Gegenstück zu *Alloc_Mem*: ein reservierter Speicherbereich wird wieder freigegeben.

Open_Screen(X, Y ,W, H, Depth, BackPen, DetailPen, Viewmodes, Name): Ptr

X (Integer): x-Koordinate der Position
Y (Integer): y-Koordinate der Position
W (Cardinal): Breite („Width") des Screens
H (Cardinal): Höhe („Height")
Depth (Integer): Anzahl der Bitplanes („Tiefe")
BackPen (Byte): Hintergrundfarbe
DetailPen (Byte): Vordergrundfarbe
ViewModes (Word): Flags (z.B. 4 für Interlace, $8000 für Hires)
Name (str): Titel

Diese Funktion eröffnet einen Screen. Die einzelnen Parameter sind identisch mit den entsprechenden Feldern in der NewScreen-Struktur.

Als „Type" wird stets 15 (Custom-screen) angenommen, „TextAttr", „Gadgets" und „CustomBitMap" werden auf Nil gesetzt. Konnte der Screen nicht geöffnet werden, bricht das Programm mit einer Fehlermeldung ab. MaxonPASCAL „merkt" sich, welche Screens mit dieser Funktion geöffnet wurden, und schließt sie ggf. am Programmende wieder automatisch.

Close_Screen (p)

p: Ptr

Dreimal dürfen Sie raten... Richtig! Screen schließen!

OpenConsole (p): Ptr

p (Ptr): Windowhandle

Diese Funktion eröffnet zu einem Window das Console.device. Zurück- gegeben wird eine Art Devicehandle. Dieser Pointer zeigt (abgesehen vom Offset 16 bzw. 64) auf den Read- bzw. Writereplyport.

ReadCon (p): Char

p (Ptr): ConsoleHandle (wird von „*OpenConsole*" zurückgegeben)

Liest vom Console.device ein Zeichen. Wurde kein Zeichen eingegeben, ist das Ergebnis chr(0). Beachten Sie aber bitte, daß viele Tasten des AMIGA-Keyboards eine ganze Folge von Zeichen liefern, die Sie dann nacheinander lesen müssen.

WriteCon (p,string)

p (ptr): Consolehandle
string (str): Zeichenkette

Das Console-device „p" (erinnern Sie sich noch? Diesen Zeiger erhalten Sie von der OpenConsole-Funktion) wird beauftragt, eine Zeichenfolge im zugehörigen Window auszugeben.

CloseConsole (p)

p: Ptr

Diese Prozedur schließt ein Console-Device, das mit „OpenConsole" geöffnet worden war.

SetStdIO (p)

p: Ptr

Die Standard-Ein-/Ausgabe wird auf das Console-Device mit der Handle „p" umgeleitet. Dadurch können Sie auf ein mit „Open_Window" geöffnetes Fenster mit den normalen E/A-Prozeduren (Read, Write, ClrScr...) zugreifen.

SetStdIO(Nil)

Aktiviert wieder die normale Standard-E/A.

Im Handbuch zu KickPascal 1.0 wurde empfohlen, beim Programmstart von der Workbench mittels

```
Reset(Input, 'CON:...');   Output:= Input;
```

Kanäle für die Standard-E/A zu definieren. Dies ist bei Version 3.0 aus Kompatibilitätsgründen weiterhin möglich. Es ist aber „sauberer", obige Befehlsfolge durch

```
Win:= Open_Window (...);
Con:= OpenConsole (Win);
SetStdIO (Con)
```

zu ersetzen.

OpenLib (var,name,version)

var: Pointer-Variable
name (Str): Name der Library
version (LongInt): Versionsnummer

Diese Prozedur entspricht der Exec-Funktion „*OpenLibrary*". Die Unterschiede:

- ☞ Der Befehl ist nicht als Funktion, sondern als Prozedur realisiert. Die Basisadresse der Library wird dann mittels VAR-Parmeter zurückgegeben.
- ☞ Kann die Library nicht geöffnet werden, wird mit einer entsprechenden Fehlermeldung abgebrochen.
- ☞ Wie inzwischen schon fast gewohnt, wird die Bibliothek am Ende ordnungsgemäß wieder geschlossen.

CloseLib(p)

p (Ptr): Zeiger auf Library

Eigentlich ist dieser Befehl überflüssig, denn, wie oben erwähnt, werden Librarys automatisch geschlossen. Trotzdem wurde er der Vollständigkeit halber implementiert.

Wait_Port(p): Ptr

p (Ptr): Zeiger auf Message-Port

Entspricht der Exec-Funktion „*WaitPort*".

Wait(Mask): Long

Mask (LongInt): Signalmaske

Ist absolut identisch mit der gleichnamigen Exec-Funktion.

Get_Msg(p): Ptr

p (ptr): Zeiger auf Message-Port

Ist identisch mit der Exec-Funktion „*GetMsg*".

Reply_Msg(p)

p (Ptr): Zeiger auf Message

Identisch mit der Exec-Funktion „*ReplyMsg*".

7.6 Nützliche MaxonPASCAL-Spezialitäten

Break (mask): integer

mask: Bitmaske mit 4 Bits:

Bit	Wert	Bedeutung
0	1	Ctrl-C
1	2	Ctrl-D
2	4	Ctrl-E
3	8	Ctrl-F

MaxonPASCAL bietet Ihnen die Möglichkeit, Programme mit der Taste F10 abzubrechen. Das ist aber nicht sehr sauber: Das Laufzeitsystem fragt hier, um nicht mehr Zeit, als unbedingt nötig zu verlieren, direkt die Tastatur ab.Deshalb wird überhaupt nicht getestet, in welchem Fenster die Taste gedrückt wurde, so daß man damit eventuell mehrere Tasks gleichzeitig unterbricht. Aus diesem Grund ist diese Möglichkeit eher als Paniktaste während der Programmentwicklung gedacht - einem Anwender sollten Sie ein solches Programm nicht zumuten.

Langer Rede schwacher Sinn: Mit der Funktion „*Break*" können Sie die Tastenkombinationen ^C und ^D abfragen (und auch noch ^E und ^F - aber es ist nicht üblich, diese Tasten zu benutzen). Die Function „*Break*" gibt „true" zurück, wenn eine der in der Bitmaske enthaltenen Taste gedrückt wurde. An geeigneten Stellen Ihres Programms sollten Sie die ^C-Taste abfragen („IF break(1) THEN...") und dann kontrolliert (geöffnete Dateien schließen, reservierten Speicher freigeben...) das Programm beenden. Spätestens wenn Sie eine lauffähige Objektdatei abspeichern, sollten Sie die „Unterbrechen"- Option des Compilers ausschalten.

CBreak

Testet, ob die Tastenkombination <Ctrl>+<C> gedrückt wurde, und bricht in diesem Fall mit der Meldung „BREAK" ab. Es entspricht damit der Anweisung „ IF break(1) THEN error('BREAK') „ und kann immer dann verwendet werden, wenn ein unkontrolliertes Abbrechen des Programms nicht schadet.

Nähere Erläuterungen zum Sinn dieses Befehls finden Sie bei der Beschreibung der Function „Break(n)".

FreeStack: LongInt

Gibt an, wie viele Bytes noch auf dem Stack frei sind. Man kann diese Funktion in rekursiven Programmen verwenden, um selbst Stacküberläufe abzufangen, sobald der Platz knapp wird.

Ein anderes Beispiel: Ein Programm mit großen Datenmengen.

```
Program Monstrum;

    Procedure Main;  { das eigentliche Hauptprogramm }
   Var a: Array[1..100000] of Real;
    { benötigt 4*100000 Bytes }
    Begin
    { hier steht das Programm }
    End;

Begin
  If FreeStack<410000 Then  { 400 KB für Array, 10 KB Reserve }
        Writeln('Zu wenig Stack, Babe!')
  Else
     Main
End.
```

Das Programm benötigt ein 400 KByte großes Array. Um eine Laufzeit- Fehlermeldung zu vermeiden, prüft das Hauptprogramm, ob genug Stack zur Verfügung steht. Nur falls dem so ist, wird die Prozedur „Main" aufgerufen, in der das Riesenarray lokal deklariert ist.

Random: Real

Liefert eine Zufallszahl mit 0 <= Random < 1. Der Zufallszahlengenerator benutzt ein Standardverfahren; zusätzlich wird aber auch die momentane Rasterposition des Bildschirm-Elekronenstrahls in die Rechnung aufgenommen, so daß die Folge richtig schön chaotisch wird.

Random (n): Integer

n: Integer

Liefert eine ganzzahlige Zufallszahl im Bereich von 0 bis n-1.

Randomize

Die Zufallszahlenfolge wird mit der momentanen Systemzeit initialisiert. Da in die Berechnung der Zufallszahlen sowieso ein nicht-mathematisches Element eingeht (s. o.), ist diese Prozedur bedeutungslos und wurde nur aus Kompatibilitätsgründen implementiert.

Hi (int): Integer

int: Integer

„Hi-Byte" einer Zahl: entspricht

```
Word(int) div 256
```

(da hier vorzeichenlos gerechnet wird), ist aber schneller.

Lo (int): Integer

int: Integer

„Lo-Byte": Niederwertiges Byte, genau wie „int AND $ff"

Upcase (c): Char

c: Char

Wandelt den Buchstaben „c", falls er zu den Kleinbuchstaben gehört, in einen Großbuchstaben. Natürlich werden auch Umlaute korrekt gewandelt. Beispiel:

```
Var s: String;
    i: integer;
Begin
  s:='O zerfrettelter Grunzwanzling!';
  For i:=1 to Length(s) do write(Upcase(s[i]))
End.
```

Ergebnis:

```
O ZERFRETTELTER GRUNZWANZLING!
```

Swap (int): Integer

int: Integer

Vertauscht Hi- und Lo-Byte des Parameters.

Frac (r): Real

r: Real-Zahl

Diese Funktion liefert die Nachkommastellen einer Real-Zahl.

Beispiel:

```
Frac(17.4)   =  0.4
Frac(-0.07)  = -0.07
Frac(1e+10)  =  0      (außerhalb der Rechengenauigkeit)
```

Es gilt: x = Trunc(x) + Frac(x) , falls x eine Realzahl ist, die im „LongInt"-Bereich liegt, so daß „Trunc" den richtigen Wert liefert.

Addr (x): LongInt

x: Beliebige Variable, Prozedur oder Funktion

Gibt die Speicheradresse, an der die Variable „x" liegt bzw. die Prozedur oder Funktion „x" beginnt, zurück.

Die Funktion

```
Addr(var)
```

(„var" = beliebige Variable) entspricht damit

`Long(^var)` (siehe auch Kapitel „Pointer").

Pwr10(i): Real

i: Integer-Ausdruck, -19 < i < 19

Diese Funktion liefert die Zehnerpotenz einer ganzen Zahl als Real. Man könnte dies auch ohne weiteres durch wiederholtes Multiplizieren erreichen, aber *„pwr10"* holt sich die Werte aus einer Wertetabelle und ist damit erstens schneller und zweitens vermeidet man die Fortpflanzung von Rundungsfehlern.

DbPwr10(i): Double

i: Integer-Ausdruck, -308 < i < +308

Dies ist die Entsprechung zu „Pwr10" im Double-Bereich - daher auch das „Db" am Anfang.

SizeOf(x): Long

x: Typbezeichner oder beliebige Variable

Liefert den Speicherbedarf eines Datentyps bzw. einer Variablen:

```
SizeOf(Integer) = 2
SizeOf(DosBase) = 4   ("Dosbase" ist eine Variable des Typs
                     "Ptr", siehe auch Kapitel IV).
```

Delay(n)

n: Integerzahl

Delay macht - genau wie die gleichnamige DOS-Funktion - eine Pause von n 50stel Sekunden.

Error(string)

string: Str

Das Programm steigt mit der angegebenen Fehler- (oder sonstiger) Meldung aus.

Halt (nummer)

nummer: integer

Programmabbruch mit der angegebenen Fehlernummer.

AddExitServer (proc)

proc: globale Prozedur ohne Parameterliste

In einem Multitasking-System wie dem AMIGA ist es oft wichtig, daß Programme nicht unkontrolliert abbrechen, sondern zuerst noch irgendwie „aufräumen", z. B. Dateien schließen, Speicher freigeben usw. Hat man über MaxonPASCAL-Prozeduren auf derartige Resourcen zugegriffen (z. B. *„Open_Window"*, *„Reset"*, *„Alloc_mem"*...), werden die entsprechenden Schließ- oder Freigabeprozeduren am Programmende automatisch ausgeführt. Wenn man aber Betriebssystemfunktionen direkt aufgerufen, z. B. mit *„AllocBitmap"* (aus der graphics.library) eine Bitmap reserviert hat, muß man selbst für ein ordentliches Programmende sorgen.

Natürlich kann man die entsprechenden Anweisungen einfach am Ende des Hauptprogramms aufrufen. Dann werden sie aber nur bei einem normalen Programmende ausgeführt, nicht beim Abbruch mit einer Fehlermeldung.

Nun stelle man sich folgendes Szenario vor: ein Programm reserviert sich mittels *„AllocBitmap"* etliche große Bitmaps. Anschließend versucht es noch, mittels der MaxonPASCAL-Funktion *„Open_Screen"* einen Screen zu öffnen, beispielsweise in der Absicht, auf den (unsichtbaren) Bitplanes Bilder einer animierten Grafik zu berechnen und sie dann auf den (sichtbaren) Screen zu blittern. Nun sei das Chip-RAM durch die *„AllocBitmap"*-Aufrufe aber schon so voll, daß kein Screen

mehr geöffnet werden kann - das Programm steigt mit der Meldung „Intuition error" aus, ohne die Bitmaps wieder freizugeben. Ergebnis: das Chip-RAM ist fast voll belegt und kann nur mit einem Reset des Systems wieder freigegeben werden.

Hier hilft die Prozedur *„AddExitServer"*: Mit ihr kann man Prozeduren angeben, die beim wie auch immer gearteten Ende eines Programms ausgeführt werden sollen. Durch wiederholten Aufruf von *„AddExitServer"* können beliebig viele solcher Exit-Prozeduren gewählt werden, die dann am Programmende in umgekehrter Reihenfolge ausgeführt werden. Die Prozeduren müssen global (also nicht lokal innerhalb einer anderen Prozedur oder Funktion definiert) sein und dürfen keine Parameter besitzen.

Beispiel:

```
PROGRAM Exitus;

  PROCEDURE Ausstieg;
     BEGIN
       Writeln('ExitServer ausgeführt.')
     END;

  BEGIN
     AddExitServer(Ausstieg);
     Writeln('Division durch 0: ', 1/0);
     Writeln('Diese Zeile wird nie ausgegeben.')
   END.
```

Das Programm erzeugt folgende Ausgabe:

```
Division durch 0:                <- "normale" Ausgabe des Programms
Division by zero.                <- Fehlermeldung des Laufzeitsystems
ExitServer ausgeführt.           <- Ausgabe des ExitServers
```

Bei jedem Aufruf der Funktionen *„Open_Window"*, *„Open_Screen"* und *„OpenConsole"* und *„OpenLib"* installiert das PASCAL-Laufzeitsystem im Prinzip nichts anderes als eine Exitserver-Prozedur, die die jeweilige Resource schließt. Wenn Sie also in Ihrem Programm zuerst einen Exitserver installieren und dann ein Window öffnen, wird am Ende des Programms erst (Aufruf der Exitserver in umgekehrter Reihenfolge!) das Fenster geschlossen und dann erst Ihre Exit-Prozedur aufgerufen. Sie können dann z. B. keine Ausgaben in das Fenster mehr machen.

Mit *„New"* oder *„Alloc_Mem"* reservierter Speicher wird dagegen erst nach der Abarbeitung aller Exitserver freigegeben (falls nicht schon geschehen), und noch geöffnete Dateien werden erst dann geschlossen.

Exit

Diese Prozedur verläßt den aktuellen Block. Wenn EXIT im Hauptprogramm aufgerufen wird, entspricht es einem GOTO-Sprung ans Programmende, in einer Prozedur oder Funktion entsprechend einem Sprung an deren Ende.

Beispiel:

```
PROGRAM Nanu;
  PROCEDURE p;
    BEGIN
      Writeln('P1');
      Exit;
      Writeln('P2')
    END;
  BEGIN
    Writeln('Start');
    p;
    Writeln('Mitte');
    Exit;
    Writeln('Ende.')
  END.
```

Das Programm erzeugt die Ausgabe:

```
Start
P1
Mitte
```

Exchange (var1, var2)

var1, var2: Variablen

Vertauscht den Inhalt zweier Variablen. Die Variablen müssen skalar sein, d.h. numerisch, Bool, Char, Aufzählungs- und Pointertypen.

Ferner müssen die Typen der Variablen kompatibel sein, insbesondere dieselbe Länge haben. Beispiel:

```
Var a, b: integer;
  Begin
    a:=26731; b:=4711;
    Exchange(a,b);
    write(a:8, b:8)
  End.
```

gibt aus:

```
    4711    26731
```

Inc (v)

v: Variable eines geordneten Typs (z. B. Ganzzahl, Char...)

Der Wert der Variablen „v" wird um 1 erhöht. Die Anweisung entspricht damit

```
v:= Succ(v),
```

Allerdings findet bei „Inc" keine Prüfung auf arithmetischen Überlauf oder Bereichsüberschreitung statt.

Beispiele:

```
{ Variablendeklaration: VAR c: Char; i: integer; b: Byte }

  c:= 'X'; Inc(c); Write(c)     gibt aus:       Y
  i:=  7 ; Inc(i); Write(i)     gibt aus:       8
  b:= 255; Inc(b); Write(b)     gibt aus:       0  (Überlauf!)
```

Dec (v)

v: Variable eines geordneten Typs (z. B. Ganzzahl, Char...)

Dies ist das genaue Geggenteil von „Inc": die Variable wird, ohne Bereichsunterschreitung oder Unterlauf abzufangen, um einen Wert erniedrigt.

7.7 Stringbehandlung

Concat(string1, string2, ..): String

Diese Funktion hängt beliebig viele Strings zusammen und gibt das Ergebnis als temporären String zurück. Diese Funktion ist im Prinzip überflüssig, da man Strings auch mit „+" aneinander hängen kann.

Copy(string, pos, len): String

String	Stringausdruck
pos:	ganzzahlige Konstante
len:	Länge (nicht-negativ)

Liefert den „len" Zeichen langen Ausschnitt aus „string" ab Position „pos".

Beispiel:

```
writeln(Copy('O zerfrettelter Grunzwanzling',6,7))
```

gibt „frettel" aus. Das Ergebnis ist übrigens stets ein temporärer String.
Falls „len" negativ ist, wird ein Fehler gemeldet.

Pos (string1, string2): integer

string1, string2: Stringvariablen oder -konstante (keine temporären Strings!)

Diese Funktion sucht, ob der erste String im zweiten vorkommt, und liefert die Anfangsposition.

Beispiele:

```
Pos('a' ,'Hallo')                  = 2
Pos('mi' , 'Du mich auch!')        = 4
Pos('ha' , 'Hallo hallo')     = 7, denn Groß-/Kleinschreibung wird
             beachtet
Pos('' , 'Yxlegrümpf')      = 1
Pos('Himpel','Forever!')       = 0, denn "Himpel" kommt in "forever!"
             nicht vor.
```

StrLen(String): integer

String: Zeichenkette (String[n], Str, Stringkonstante, Char...)

Ermittelt die Länge des übergebenen Strings (also die Anzahl der Zeichen vor dem ersten Null-Char).

Length (String): integer

Ist identisch mit *„StrLen"*.

Insert (string, stringvar, pos)

string: String-Ausdruck
stringvar: Variable eines STRING[n]-Typs
pos: ganzzahlige Position, pos >= 1

In eine Stringvariable wird an einer bestimmten Stelle ein Stringausdruck eingefügt. Falls die Position nicht positiv ist, außerhalb der bisherigen Stringlänge liegt oder falls das Ergebnis zu lang für die Stringvariable ist, wird ein Fehler gemeldet.

Beispiel:

```
VAR st, name: STRING;
BEGIN
   st:= 'Hallo!';
   Write ('Name: '); ReadLn (Name);
   Insert (' '+Name, st, 6);
   Writeln (st)
END.
```

Dieses Programm fügt den eingegebenen Namen und ein Leerzeichen vor dem „!" in die Variable „st" ein und gibt dann also „Hallo ...!" aus.

Delete (stringvar, pos, len)

stringvar: Variable eines STRING-Typs
pos: ganzzahlige Position, pos >= 1
len: ganzzahlige Länge, len >= 0

Aus einer Zeichenkette werden ab einer bestimmten Postion eine bestimmte Anzahl von Zeichen gelöscht.

Beispiel:

```
VAR st: STRING;
BEGIN
   st:= 'AMIGOXYA';
   Delete (st, 5, 3);
   WriteLn (st)
END.
```

In diesem Programm werden die Buchstaben „OXY" aus dem String gelöscht.

Ergebnis:

```
AMIGA
```

RealStr(float, digits): String

float: Real-Ausdruck
digits: Stellenanzahl (ganzzahlig)

Wandelt die Real-Zahl in eine Zeichenkette und gibt diese zurück, und zwar exakt die Zeichenfolge, die mit „Write(float:0:digits)" ausgegeben würde. Näheres zur Bedeutung des zweiten Parameters steht im Kapitel über die Numerischen Datentypen. Die Zeichenkette, die „RealStr" zurückgibt, ist temporär (siehe auch Kapitel „Stringtypen").

IntStr (i): tempString

i: LongInt-Ausdruck

Aus dem Ganzzahl-Ausdruck wird die entsprechende Dezimalzahl erzeugt und als temporärer String zurückgegeben.

Beispiel:

```
InsStr(17)      ergibt     '17'.
```

Val (string, numvar, intvar)

string: String-Ausdruck
numvar: Variable des Typs Real, Integer oder LongInt
intvar: Variable des Typs Integer oder Cardinal/Word

Diese Prozedur versucht, den Stringausdruck „string" in eine Zahl zu wandeln, und zwar abhängig vom Typ der Variablen „numvar" in eine Real-, Double-, Integer- oder LongInt-Zahl. Ist der String eine korrekte dezimale Zahldarstellung (ohne führende oder folgende Leerzeichen!), so erhält „numvar" den entsprechenden Zahlenwert und „intvar" den Wert 0. Falls der String nicht korrekt gewandelt werden konnte, ist „numvar" undefiniert und „intvar" enthält die Nummer des ersten Zeichens des Strings, das nicht paßt.

Beispiele:

```
Var R:Real; L:Long; I,p:Integer; ...

Val('4711', R, p)        ==> R=4.711E+4, p=0
Val('-5', L, p)          ==> L=-5,       p=0
Val('50000', I, p)       ==> I=undef.,   p=6 (weil 50000 > MaxInt)
Val('+3.14149', L, p)    ==> L=undef.,   p=3  (keine ganze Zahl)
```

8. Compiler-Anweisungen, Includefiles und Betriebsystemfunktionen

8.1 Compiler Directives

8.1.1 Includefiles

Es gibt in PASCAL eine Klasse von Anweisungen, die nicht zum Programm gehören, sondern zur Steuerung des Compilers bestimmt sind. Sie sind leider nicht genormt. Es ist aber üblich, sie als Kommentare zu tarnen:

```
{$Anweisung}   oder   (*$Anweisung*)
```

Wie Sie sehen, sind diese Compiler Directives äußerlich nichts anderes als Kommentare, deren erstes Zeichen ein „$" ist. Der Compiler wird dadurch angewiesen, diesen Kommentar nicht etwa wie gewohnt zu überlesen, sondern die darin enthaltene Anweisung sofort auszuführen.

Aber gehen wir doch von der Theorie gleich zum ersten Beispiel:

```
{$incl "Dateiname" }
```

Diese Sequenz weist den Compiler an, eine Includedatei einzuladen.

„Was ist das?"

Es ist nicht immer sinnvoll, den gesamten Quelltext im Editor zu haben. Wenn Sie z.B. einige Prozeduren bereits fertiggestellt haben, können Sie diese in einer Textdatei ablegen und aus Ihrem Programm löschen, wodurch dieses kürzer und handlicher wird. Im Rest Ihres Programms weisen Sie den Compiler an der Stelle, wo vorher der ausgelagerte Teil gestanden hat, mit einer Directive an, diesen Teil als Includedatei zu laden. Er tut dann so, als würde der Inhalt der Datei an dieser Stelle im Quelltext stehen, und compiliert so, als wäre nichts geschehen.

Zur Programmierung des Betriebssystems stehen Ihnen einige fertige Hilfsdateien zur Verfügung, die Sie auf diese Weise in Ihr Programm übernehmen können. Doch dazu später mehr.

Alles, was hinter einer Directive steht, wird wie ein gewöhnlicher Kommentar überlesen:

```
{$incl"Hallo"  Diese Anweisung öffnet die Includedatei "Hallo" }
```

Sie können aber auch mehrere Directives hintereinander hängen, indem Sie sie mit einem Semikolon „;" trennen bzw. verbinden (kommt auf den Standpunkt an):

```
{$incl "Hallo";Incl"Horrido" ; include"Hallali" }
```

Wie Sie hier ebenfalls sehen, dürfen Sie statt „incl" auch „include" oder irgendetwas anderes schreiben - vorausgesetzt, es fängt mit „incl" an. Ferner zeigt obiges Beispiel, daß die Groß- und Kleinschreibung bzw. das Einschieben von Leerzeichen dem Compiler egal sind.

Dieselbe Wirkung wie oben hätten Sie übrigens auch einfacher haben können:

```
{$incl "Hallo","Horrido","Hallali"}
```

Die Sache hat allerdings noch einen Haken: die Dateien werden aus dem aktuellen Verzeichnis geladen. Sie hätten natürlich auch einen ganzen Pfad angeben können, z. B. so:

```
{$incl "PASCAL:Includedateien/Gruß/Hallo" }
```

Wenn Sie aber mehrere solcher Dateien einladen wollen, wird es lang und umständlich. Zudem müßten Sie, wenn das Verzeichnis einmal wechselt (z. B. weil Sie auf der Diskette aufgeräumt oder sie umbenannt haben), alle Dateinamen ändern. Dagegen hilft eine andere Directive: PATH. Unser Beispiel sähe dann so aus:

```
{$path"PASCAL:Includedateien/Gruß/"; incl"Hallo","Horrido",Hallali"}
```

Dieses „path" gibt also einen alternativen Pfad an, über den Includedateien gesucht werden. MaxonPascal sucht Includedateien immer zuerst im aktuellen Verzeichnis und dann über diesen Pfad. Die Directive „$path" ist aber ziemlich überflüssig, denn man kann den Suchpfad für Includes auch über den Compiler-Requester einstellen. Default ist „pascal:include".

Übrigens: Sie können Includedateien auch verschachteln, d.h. eine Datei kann selbst wieder andere einladen.

Nun gibt es noch ein weiteres Problem: Der Compiler ist zu schnell! Er ist so unverschämt schnell, daß die Floppy beim Compilieren von Includefiles nicht mitkommt und der Compiler auf sie warten muß. Das ist natürlich Zeitverschwendung - zwar eine, mit der man leben kann, aber trotzdem ärgerlich. Deshalb gibt es bei der „Path"-Directive eine besondere Option: Sie können zwei verschiedene Pfade, durch ein Komma getrennt, angeben, z. B. so:

```
{$path "ram:include","pascal:include" }
```

Die Wirkung ist folgende: Bei „incl" wird die Datei zuerst im ersten Verzeichnis (hier: „ram:include/") gesucht. Ist sie dort vorhanden, wird sie auch von dort gelesen. Falls dem aber nicht so ist, holt der Compiler die Includedatei über den zweiten Pfad. Und jetzt kommt der raffinierte Trick: Sie wird nicht nur einfach gelesen, sondern gleichzeitig in das erste Verzeichnis kopiert, so daß sie beim nächsten Compilerlauf dort zur Verfügung steht! Auf diese Weise können Sie sich ganz einfach und automatisch die Dateien, die Sie benötigen, von der langsamen Diskette in die schnelle RAM- oder RAD-Disk kopieren...

Und es wird noch schöner: Beim Kopieren legt MaxonPASCAL bei Bedarf von selbst die Unterverzeichnisse an, sofern sie noch nicht existieren! Im obigen Fall würde also immer zuerst geprüft, ob das Directory „ram:include" bereits existiert. Zu diesem Thema noch ein Beispiel:

```
{$path "ram:include/","pascal:include/"}
{$incl "exec/tasks.h","exec/ports.h","devices/trackdisk.h"}
```

Hier werden drei der AMIGA-System-Includedateien eingeladen. Beim ersten Compilerlauf, wenn die Ramdisk noch leer ist, liest MaxonPASCAL die Dateien aus dem Verzeichnis „PASCAL:include", legt in der Ramdisk mehrere verschachtelte Unterverzeichnisse an (erst „include" und dann darin die Unter-Unter-Directories „exec" und „devices") und kopiert die vier Dateien dahin. Bei allen folgenden Compilierungen kann MaxonPASCAL dann wesentlich schneller auf die Includefiles zugreifen, so daß sie jeweils in Bruchteilen einer Sekunde übersetzt werden.

8.1.2 Bedingte Compilierung

Weitere Directives sind IF, ELSE und ENDIF. Sie erlauben Ihnen, den Compiler zu steuern. Hinter der IF-Directive ist eine Bedingung anzugeben, evtl. mit vorangestlltem „not". In der vorliegenden Version ist aber erst eine einzige derartige Bedingung implementiert: „DEF ident". Diese Bedingung ist wahr genau dann, wenn der dahinter angegebene Bezeichner definiert ist.

Ein Beispiel:

```
Program CompilerDirectives;
Const Deutsch=26731;  { Löschen, wenn Englisch erwünscht }
Begin
  {$if def deutsch}
  writeln('Hallo');
  {$else}
  writeln('Hello');
  {$endif}
  {$if def yxlegrümpf}
  Dieses hier wird immer überlesen.
  {$endif}
End.
```

Auf diese Weise können Sie Ihr Programm in zwei Sprachen gleichzeitig schreiben. Je nach dem, ob die Konstante (Sie könnten aber auch eine Variable, einen Typen, eine Prozedur... nehmen) „Deutsch" definiert ist, erfolgt eine deutsch- oder englischsprachige Ausgabe. Beachten Sie aber den gewaltigen Unterschied zur gewöhnlichen IF-Then-Else-PASCAL-Struktur: Dabei fällt die Entscheidung während der Laufzeit des Programms, hier schon während des Compilierens. Ferner wird hier das, was ausgeschlossen wird, vom Compiler völlig ignoriert, so daß es nicht einmal syntaktisch richtig sein muß.

Eine weitere Anwendung finden Sie in den meisten der mitgelieferten Includefiles. Unter diesen Dateien herrscht eine gewisse Hierarchie: einige benötigen selbst wieder andere Includedateien, um vom Compiler verstanden zu werden. Bevor diese anderen Includes eingeladen werden, muß aber geprüft werden, ob der Compiler sie nicht vielleicht schon längst geladen hat, denn sonst würde es zu Fehlermeldungen wegen Doppeldeklarierungen kommen. Deshalb enthält z. B. die Datei „exec/devices.h" diese Zeilen:

```
{$if not def EXEC_DEVICES_H }
Const EXEC_DEVICES_H=0;
{$if not def EXEC_LIBRARIES_H; incl"exec/libraries.h";endif}
```

```
{$if not def EXEC_PORTS_H; incl"exec/ports.h";endif}
```

...Hier kommt der eigentliche Inhalt der Datei ...

```
{$endif}
```

Um die ganze Datei legt sich hier eine IF-ENDIF-Klammer, die sichert, daß diese Datei wirklich nur einmal vom Compiler übersetzt wird. Um dies feststellen zu können, wird innerhalb dieser Konstruktion eine Konstante namens „exec_devices_h" definiert. Ferner benutzt die Datei Bezeichner, die in den Dateien „exec/libraries.h" und „exec/ports.h" deklariert werden. Auch diese Dateien deklarieren Bezeichner, die wie die jeweilige Datei heißen. Damit kann der Compiler prüfen, ob er diese Dateien schon übersetzt hat, und sie nur dann öffnen, wenn dies nicht längst geschehen ist.

8.1.3 Error

MaxonPASCAL hat noch einige weitere Directives auf Lager, z. B. „Error". Damit können Sie sich individuelle Fehlermeldungen ausgeben lassen.

Eine denkbare Anwendung wäre z.B., wenn zwei Includedateien sich gegenseitig ausschließen, weil es zu Doppeldeklarierungen kommen würde. Ein Beispiel: Sie wollen ein Programm sowohl in einer deutsch- als auch in einer englischsprachigen Version schreiben. Dazu deklarieren Sie alle Texte in zwei verschiedenen Dateien als String-Konstanten. Die Datei „Deutsch.texte" könnte dann etwa so aussehen:

```
{$if def Englisch_texte;
  Error "Sie können nur eine Sprache verwenden";
  else }
 Const
  Deutsch_texte= 0;
  Gruß          = 'Guten Tag!';
  Abschied      = 'Auf Wiedersehen!';
{$endif}
```

„Englisch.texte" wäre dann genau anders 'rum:

```
{$if def Deutsch_texte;
  Error"Sie können nur eine Sprache verwenden";
  else }
 Const
  Englisch_texte= 0;
  Gruß          = 'Hello!';
  Abschied      = 'Bye!';
{$endif}
```

Bei zwei so einfachen Includedateien ist es natürlich kein Problem, auch ohne derartige Fehlermeldungen die Übersicht zu behalten. Wenn Sie aber einen ganzen Haufen ineinander verschachtelter Dateien anlegen, kann dies recht nützlich werden. Denn wenn in einer Includedatei ein

Fehler auftritt, setzt der Compiler den Editor- Cursor nicht etwa auf die fehlerhafte Stelle (die ja gar nicht im Arbeitsspeicher, sondern in einer Datei liegt), sondern auf die zugehörige „incl"-Anweisung.

8.1.4 Linkersteuerung

Mit der Compiler-Anweisung

```
{$link "datei1", "datei2", ... , "dateix" }
```

wird der Linker angewiesen, die angegebenen Dateien zu laden und zum compilierten Programm hinzuzulinken. Die Dateien werden, wenn der Compiler auf diese Directive stößt, sofort geladen; gelinkt wird aber erst nach dem erfolgreichen Compile-Vorgang.

Die Directive darf, wie alle anderen auch, an jeder Stelle des Quelltextes stehen.

Im Gegensatz zu „incl" gibt es zu „link" keine Pfad-Anweisung und auch keine Umkopier-Möglichkeit.

Die Directive „ulink" arbeitet wie „link", holt die Datei aber aus dem in der Config-Datei angegebenen Unit-Verzeichnis. Im Make-Modus werden "$link" und "$unlink" grundsätzlich ignoriert, da hier die Projektverwaltung für das Linken verantwortlich ist. Alles weitere zu dieser Directive finden Sie im Kapitel 9 „Units und Module".

8.1.5 Compiler-Optionen

Mit dem Pull-Down-Untermenü **„Optionen/Compiler"** können Sie einige Compiler-Optionen einstellen. Der Nachteil: Diese Einstellungen wirken nur global, gelten also für das gesamte Programm, und es geht nicht aus dem Quelltext hervor, welche Einstellung sinnvoll ist, so daß man immer, wenn man einen Quelltext in den Editor lädt, die Optionen von Hand einstellen muß.

Deshalb gibt es eine Compiler-Directive, mit der man diese Einstellungen vornehmen kann. Sie heißt

```
{$opt para1, para2, ... , paraX }
```

Syntax für einen einzelnen Parameter:

```
(("t"|"i"|"b"|"s"|"a") ("+"|"-"|"0")) | "q"
```

Die Anfangsbuchstaben der Parameter entsprechen den Tasten im Tastatur-Untermenü „Preferences" und wirken auch so:

{$opt q }	schaltet alle Optionen aus („Quick")
{$opt x+ }	die Option „x" (hier ist natürlich „t", „i", „b", „s" oder „a" einzusetzen) wird eingeschaltet - entspricht „ON" im Tastaturmenü oder einem abgehakten Punkt im Pull-Down-Menü.
{$opt x- }	schaltet die Option „x" aus („x" wie oben zu ersetzen).

{$opt x0 } die entsprechende Option wird auf den Wert geschaltet, der im Menü eingestellt ist. Dies ist natürlich nur sinnvoll, wenn die Option vorher mit „x+" oder „x-" ein- oder ausgeschaltet wurde.

Beispiele:

```
{$opt q,s+}
```

schaltet alle Optionen außer dem Abfangen eines Stacküberlaufs aus.

```
s:= 0;
{$opt b-,a-}
FOR i:=1 to 1000 dO s:= s + a[i];
{$opt b0,a0}
```

In diesem Programmfragment werden die 1000 Elemente eines Arrays summiert. Um dies zu beschleunigen, werden vor der Schleife die Unterbrechungsmöglichkeit (mit Taste <F10>) und die Prüfung auf Überlauf ausgeschaltet. Nach der Schleife verwendet der Compiler wieder die Einstellung des Menüs.

Die Optionen gelten also von der Stelle an, wo die „opt"-Directive steht, bis sie durch eine folgende Directive wieder anders gesetzt werden. Am Anfang, also vor einer „opt"-Directive, verwendet der Compiler natürlich wie gewohnt die im Menü eingestellten Optionen.

Nun aber das Wichtigste: Die Bedeutung der einzelnen Optionen.

t - Subrange testen

Bei Wertzuweisungen an als Ausschnittstyp deklarierte Variablen ist zur Laufzeit der Wertebereich zu prüfen. Aber man benutzt Ausschnittsypen ja gerade, um solche Bereichsüberprüfungen durchführen zu lassen, so daß man diese - defaultmäßig aktivierte - Option wohl nur ausschalten wird, wenn man eine Exe-Datei eines ausgetesteten Programms erzeugen lassen will.

Bei Zuweisung konstanter Werte wird der Bereich schon zur Compile-Zeit geprüft, und zwar unabhängig von der „t"-Option. Das folgende Programm wird also immer eine Compiler-Fehlermeldung verursachen:

```
Var n: 1..100;
Begin n:= 101 End.
```

Dagegen findet bei folgendem Programm die Prüfung zur Laufzeit statt, und zwar nur dann, wenn die Option „t" aktiv ist:

```
Var n: 1..100;
Begin
  n:= 100;  { im erlaubten Bereich }
  n:= n+1;  { Bereichsüberschreitung, erst zur Laufzeit feststellbar }
End.
```

i - Indexbereich bei Array-Zugriff

Diese Option sorgt dafür, daß bei jedem Zugriff auf ein Array-Element getestet wird, ob der Index im richtigen Bereich liegt. Man sollte diese Option nur ausschalten, wenn das Programm wirklich durchgetestet und debuggt ist, denn die Folgen können sonst gleichermaßen fatal (GURU!) wie heimtückisch (Fehler schwer zu finden) sein: Zwar liefert ein Lesezugriff auf ein nicht-existentes Feldelement nur einen undefinierten Wert, aber eine derartige Wertzuweisung schreibt einen Wert irgendwo in den Speicher. Meist (jedenfalls bei geringen Abweichungen vom deklarierten Indexbereich) liegt an dieser Stelle eine andere Variable, die dann aus scheinbar unerklärlichen Gründen einen anderen Wert erhält - viel Spaß bei der Fehlersuche...

Es kann aber auch passieren, daß Sie das System zerschießen und Ihr AMIGA wieder 'mal nach Indien pilgert. Also Vorsicht beim Ausschalten dieser Option!

Beispiel:

```
{$opt i-}
Var a: Array[1..10] of integer;
    k: integer;
Begin
  k:= 42;
  a[0]:= 26731;    { FALSCHER INDEX!!! }
  writeln(k)
End.
```

Dieses Programm gibt „26731" aus.

b - Unterbrechen

Es ist ausgesprochen ärgerlich, wenn ein Programm während eines Probelaufs unaufhaltbar in einer Endlos-Schleife hängenbleibt. Zwar kann man dann den Quelltext problemlos speichern, aber man muß neu booten, um das Programm zu stoppen. Deshalb gibt es eine Taste zum Abbrechen von PASCAL-Programmen - nämlich die Funktionstaste <F10>. Wenn die Option „b" aktiv ist, baut der Compiler an geeigneten Stellen Tastaturabfragen in den Code ein, und zwar überall dort, wo eine Endlosschleife entstehen könnte:

- Am Anfang von WHILE- und REPEAT-Schleifen
- Bei Labels
- Innerhalb jeder FOR-Schleife (die zwar theoretisch nicht endlos, praktisch aber ganz schön lang sein kann)
- Am Anfang von Unterprogrammen (endlose Rekursion!)

Um nicht mehr Zeit als unbedingt nötig zu brauchen, fragt das Programm die Tastatur hardwaremäßig ab - auf einem Multitasking-System gewiß nicht das Gelbe vom Ei, denn stellen Sie sich einmal vor, Sie lassen Ihr mit MaxonPASCAL entwickeltes Apfelmännchen-Programm im

Hintergrund rechnen, während Sie schon das nächste Programm entwickeln und dabei im MaxonPASCAL-Editor mit der Taste <F10> ans Textende springen...

Sie sehen also, daß die Abbruchmöglichkeit mit <F10> nur als Notausstieg während der Programmentwicklung taugt. Wenn Sie ein Programm schreiben, das der Benutzer abbrechen können soll, ist es wohl besser, an geeigneten Stellen (zum Beispiel in Schleifen - siehe oben!) mit dem „*CBreak*"-Befehl oder der Funktion „Break(1)" die Tastenkombination „ ^ C" abzufragen.

Wenn Sie ein Programm geschrieben haben, das länger rechnen soll (ein paar Minuten, über Nacht oder so), ist es oft keine leichte Entscheidung, ob man die Breakpoints setzen soll. Wenn Sie sie rauslassen, riskieren Sie, Ihr Programm nur noch durch einen Reset abbrechen zu können. Der Laufzeitbedarf der Tastenüberprüfung ist unterschiedlich, macht sich aber vor allem bei engen Schleifen bemerkbar: Die Laufzeit einer leeren FOR-Schleife steigt dadurch um ca. 75%, wenn die Schleife aber nennenswerte Anweisungen enthält, ist der Laufzeitaufwand eher marginal.

Beispiel:

```
VAR L,S:Long;
BEGIN
   {$opt q,b0}
   S:= 0;
   FOR L:=1 to 200000 DO S:= S + (L-10000)
END.
```

Bei dieser kleinen Schleife steigt die Laufzeit durch die Abbrechmöglichkeit nur noch um ca. 25%, und wenn der Schleifenkörper mehr Anweisungen enthält (dürfte die Regel sein), ist der Zeitaufwand für die Tastaturabfrage im Vergleich zu den restlichen „Kosten" eines einzelnen Schleifendurchlaufs lächerlich gering.

s - Stacküberlauf abfangen

Auf dem Stack (Stapelspeicher) werden zur Laufzeit die nicht als STATIC deklarierten Variablen abgelegt (lesen Sie dazu bitte auch den Abschnitt „dynamische und statische Variablen" im Kapitel „Module").

Die Option „s" sorgt nun dafür, daß dabei stets auch geprüft wird, ob noch genug Speicher frei ist - andernfalls nimmt das Programm sich den Speicher einfach, egal, ob was frei ist oder nicht.

Auch diese Compiler-Option benötigt etwas (aber nicht viel) Laufzeit. Vor allem (falls überhaupt) macht sich die Verzögerung bei rekursiven Prozeduren bemerkbar - aber gerade dort ist sie eigentlich unverzichtbar, denn der Stackbedarf von Rekursionen läßt sich nur schwer abschätzen und wird auch nicht bei „Linear Stack Requirement" berücksichtigt.

a - Überlauf bei arithmetischen Operationen melden

Bei arithmetischen Operationen wird jeweils auf Über- oder Unterlauf geprüft. Dies kostet vor allem bei Additionen und Subtraktionen von Integer-Zahlen (aller Längen, versteht sich) Laufzeit. Bei dem

folgenden Programm erhöht es beispielsweise die Rechenzeit um etwa 25 Prozent. Bei Multiplikationen oder sämtlichen REAL-Operationen ist der Effekt nicht so gravierend, da diese Operationen ohnehin ewig dauern (nach Computer-Maßstäben).

```
VAR i,j,k,l,m:integer;
BEGIN
   {$opt q,a0}
   l:= 0;
   m:= 0;
   FOR i:=1 TO 50 DO
     FOR j:= 1 TO 50 DO
       FOR k:= 1 TO 100 DO
         BEGIN
           l:= i+j-k
           m:= l+k
         END;
END.
```

Achtung: Bei LongReal-Rechnungen kommt es nicht zu einer Fehlermeldung, sondern zu einer Exception, und zwar unabhängig von dem „a"-Compilerswitch!

8.1.6 Kurzauswertung boolescher Ausdrücke

Werten Sie doch einmal den folgenden logischen Ausdruck aus:

```
((1 < 0) and (2 < 3))  or  ((1 < 2) or (1 = 2))
```

Falls Sie ein wenig Erfahrung mit boolescher Algebra haben, werden Sie gar nicht alle Terme des Ausdrucks betrachten, sondern etwa so vorgehen: Der linke Teil der AND-Verknüpfung ist FALSE, also ist das Ergebnis unabhängig von der rechten Seite FALSE. Bei der zweiten OR-Verknüpfung ist der erste Term TRUE, also auch das Ergebnis dieses OR. Das Gesamtergebnis des Ausdrucks ist also TRUE.

Sinnvollerweise kann MaxonPASCAL logische Operanden genauso auswerten: Ist der erste Operand von AND logisch falsch oder der erste von OR wahr, wird der zweite Operand gar nicht erst betrachtet, denn der hat auf das Ergebnis keinen Einfluß mehr.

MaxonPASCAL 1.0 wertete noch immer beide Operanden aus, bevor es das Ergebnis berechnete. Aus Gründen der Kompatibilität gibt es in Version 3 aber einen Umschalter, mit dem man eine Voll-Auswertung erzwingen kann. Er wurde in Form einer Compiler- Directive realisiert und heißt wahlweise

```
{$bool full} oder  {$bool all}.
```

Die Einstellung gilt dann, wie gewohnt, von der Stelle an, wo die Directive steht, bis zu der Stelle, wo wieder umgeschaltet wird. Das geschieht mit

```
{$bool min} oder  {$bool fast},
```

ganz wie Sie wollen.

Jetzt stehen Sie möglicherweise wieder an der Stelle, wo Sie ausrufen: „Was soll das?", oder vielleicht auch: „Cui bono?" Denn da es ja klar ist, daß die Kurzauswertung Zeit spart, hat man nicht das Bedürfnis, in einen langsameren Modus zu schalten. Oder?

Natürlich hat es auf das Laufzeitverhalten eines Programms keinen Einfluß, ob ein Teilausdruck wie „2<3" ausgewertet wird oder nicht.

Allerdings können Sie in booleschen Ausdrücken auch Funktionen mit Seiteneffekt betrachten, d. h. selbstdefinierte Funktionen, in deren Anweisungsteil globale Variablen verändert werden (schrecklicher Programmierstil!) oder die Ein- oder Ausgaben machen. Ein Beispiel für letztere Unart:

```
Program
Die_vogonische_Poesie_ist_die_drittschlechteste_des_Universums;

Function Frage: Boolean;
    Var c: Char;
    Begin
      write('Wollen Sie das Gedicht wirklich lesen? '); readln(c);
      Frage:= (c In ['j', 'J'])
    End;

 Function Bestätigung: Boolean;
    Var c: Char;
    Begin
      write('Sind Sie ganz sicher? '); readln(c);
      Bestätigung:= (c In ['j', 'J'])
    End;

 {$bool full - bewirkt hier fehlerhaften Programmablauf! }

Begin
   If Frage and Bestätigung Then
      writeln('Na gut, Sie haben es nicht anders gewollt:'\10\10,
      'O zerfrettelter Grunzwanzling!'\10,
      'Dein Harngedränge ist für mich'\10,
      'Wie Schnatterfleck auf Bienenstich...'\e'0;33mWü',
      ''\e'Büü'\e'Büä'\e'Bä'\e'Bä'\8\e'Bä'\8\e'Br'\8\e'Bg'\8,
      ''\e'B.'\8\e'B.'\8\e'B.')
End.
```

Jede der beiden Functions fragt den User, ob er sich das Gedicht wirklich antun will, und gibt die Antwort als booleschen Wert zurück.

Normalerweise steigt das Programm aus, wenn die erste Frage nicht mit „Ja" beantwortet wird. „Bool Full" bewirkt aber, daß dann trotzdem noch die zweite Frage gestellt und erst dann der Ausdruck ausgewertet wird. Natürlich kann man sich auch umgekehrt Programme konstruieren, die ausschließlich mit „Bool Full" korrekt laufen, aber das dürfte in der praktischen Programmierung kaum vorkommen, so daß defaultmäßig Kurzauswertung vorgenommen wird. Wenn Sie aber ein mit MaxonPASCAL entwickeltes Programm portieren wollen, sollten Sie daran denken, daß nicht jeder Compiler Kurzauswertung beherrscht - zum Beispiel die Version 1.0 von MaxonPASCAL!

8.1.7 Behandlung von Ein-/Ausgabefehlern

Die saubere Programmierung von Ein- und Ausgabeoperationen ist oft eine lästige Angelegenheit, denn hier muß das Programm mit einem Benutzer, einem Peripheriegerät oder dergl. kommunizieren, wobei es praktisch immer zu Fehlern kommen kann. Typische Beispiele sind Fehleingaben des Benutzers, der Versuch des Lesens einer nicht vorhandenen Datei usw. Wenn man ein Programm nicht nur selbst benutzen, sondern auch anderen Anwendern zugänglich machen will, muß man bei jeder nennenswerten EA-Operation dafür sorgen, daß eventuell auftretende Fehler nicht zu einem Fehlverhalten des Programms führen (OK, bei einfachen Bildschirmausgaben mit Write[Ln] dürften in etwa nie Fehler auftreten).

Der Normalfall ist, daß nach einer EA-Operation die Funktion „IOResult" eine Fehlernummer <> 0 liefert, so daß das Programm den Fehler erkennen und in geeigneter Weise darauf reagieren kann. Seit Version 2.0 gibt es aber die Möglichkeit, Code erzeugen zu lassen, der bei einem EA-Fehler mit einer Fehlermeldung aussteigt. Das ist zwar zugegebenermaßen nicht die benutzerfreundlichste Art der Fehlerbehandlung (man hat gerade 1000 Adressen in sein Dateiprogramm eingetipt und dann steigt das Ding beim Abspeichern aus...), aber wenigstens weiß der Benutzer dann, das etwas schiefgelaufen ist.

Das funktioniert, wie alles in diesem Kapitel 8.1, über eine Compiler Directive. Sie heißt wahlweise „i" (ist kompatibler) oder „io" (ist aussagekräftiger) und wird von einem „+" (Fehlerkontrolle einschalten) oder einem „-" (im folgenden Fehler ignorieren) gefolgt.

Beispiel:

```
VAR i, j: integer;
BEGIN
     Write ('Bitte zwei Zahlen eingeben: ');

     {$io+: ab jetzt bei Fehlern aussteigen }

     Read (i);

     {$io-: ab jetzt Fehler ignorieren }

     Read (j);

     IF IOResult <> 0 THEN
       Write ('Die zweite Eingabe war nicht richtig.');
     Writeln;
     Write ('Eingabe war: ', i:8, j:8)
END.
```

Bei Eingabe „x 4" steigt das Programm mit einer Laufzeitfehlermeldung aus, während es bei Eingabe „4 x" den Fehler erkennt und meldet.

Sie können sich die Funktionsweise der $io-Directive im Prinzip so vorstellen, als ob hinter jede Anweisung, die mit EA zu tun hat,

```
IF IOResult <> 0 THEN Error ('File error');
```

geschrieben würde.

Folgende Prozeduren werden von $io behandelt:

ClrScr
Get
Page
Put
Read
ReadLN
Reset
Rewrite
Seek
Write
WriteLN

Bei aktivierter $io-Option kann Ihr Programm noch nicht einmal Fehler, die beim Öffnen einer Datei auftreten, selbst abfangen. Deshalb ist es nicht sinnvoll, sie für das ganze Programm einzuschalten.

8.2 Libraries

Das Betriebssystem des AMIGA ist in mehrere Libraries aufgeteilt. Mit MaxonPASCAL können Sie alle diese Funktionen verwenden. Dazu müssen sie aber zuerst deklariert werden.

Jede Library besitzt eine Basisadresse. Unterhalb dieser Adresse liegt im Speicher stets eine Tabelle von Sprungbefehlen, die zu den jeweiligen Libraryfunktionen führen. Die Differenz zwischen der Basisadresse der Bibliothek und der Adresse des Einsprungs heißt der Offset der Funktion. Diese Offsets sind stets negativ (weil ja die Sprungtabelle, wie gesagt, unterhalb der Basisadresse liegt) und ist durch 6 teilbar (weil ein normaler Sprungbefehl in Maschinensprache 6 Bytes belegt).

Zunächst brauchen Sie also die Basisadresse der Library, die Sie benutzen wollen. Die Basen der „exec.library" und der „dos.library" stehen Ihnen durch die Variablen „SysBase" und „DosBase" zur Verfügung, bei allen anderen Bibliotheken erhalten Sie die jeweilige Basis mit Hilfe der Prozedur „Open_Library", die bereits im Kapitel „Prozeduren und Funktionen" erläutert wurde. Diese Adresse müssen Sie in irgendeiner Variablen ablegen, die vom Typ „Long"/"LongInt" oder einem Pointertypen sein kann.

Als nächstes sind die einzelnen Funktionen der Library zu definieren. Eine Library-Deklaration kann irgendwo im Deklarationsteil des Programms bzw. eines Unterprogramms stehen und hat folgende Syntax:

```
"Library" Basis ":" {Offset ":" ProcOrFuncHeading [";"]} "End;"
```

Als Beispiel betrachten wir einen Teil der „exec.library", den Sie in dem Includefile „exec/lists.h" finden. Es beginnt mit dem Kopf:

```
Library SysBase:
```

Als Basis kann man einen beliebigen Ausdruck verwenden, aber eine einfache Variable ist wohl am sinnvollsten.

Nun folgen die einzelnen Prozeduren und Funktionen. Die erste heißt „Insert" und hat den Offset -234. Da es aber zu einer Kollision mit der gleichnamigen MaxonPASCAL-Prozedur kommen würde, mußte sie umbenannt werden, und zwar in „_Insert". Das ist möglich, denn die Betriebssystemfunktion „weiß" ja überhaupt nichts davon, unter welchem Namen sie aufgerufen wurde - Hauptsache, der Offset und die Parameter stimmen.

Also sieht die Deklaration so aus:

```
-234: Procedure _Insert(a0:p_List; a1,a2: p_Node);
```

Der Teil hinter dem „:" ist ein ganz normaler Prozedurkopf. Nun weiß der Compiler, daß dieser Procedure drei Parameter zu übergeben sind, von denen der erste vom Datentyp „p_List" und die beiden anderen vom Typ „p_Node" sind (diese beiden Datentypen wurden natürlich zuvor in dem Includefile definiert). Nun müssen die Parameter aber nicht in irgendwelchen Variablen, sondern in Prozessorregistern übergeben werden.

Der Prozessor 68000 hat die acht Datenregister d0, d1, .. d7 und acht Adressregister („a0" bis „a7"). In allen diesen Registern können den Betriebssystemfunktionen Parameter übergeben werden - mit drei Ausnahmen: „a7" ist der Stackpointer, in „a6" wird unmittelbar vor dem Aufruf die Basisadresse gelegt und „a5" hat in MaxonPASCAL eine besondere interne Bedeutung. Diese drei Register dürfen also nicht verändert werden.

Zurück zum Beispiel: Es folgen noch sechs Prozeduren und eine Funktion:

```
-240:Procedure AddHead(a0:p_List; a1:p_Node);
-246:Procedure AddTail(a0:p_List; a1:p_Node);
-252:Procedure Remove(a1:p_Node);
-258:Procedure RemHead(a0:p_List);
-264:Procedure RemTail(a0:p_List);
-270:Procedure Enqueue(a0:p_List; a1:p_Node);
-276:Function FindName(a0:p_List; a1:Str):p_Node;
```

Man beendet die Library-Deklaration mit:

```
End;
```

Noch einmal das Ganze im Überblick:

```
Library SysBase:
 -234:Procedure _Insert(a0:p_List; a1,a2:p_Node);
 -240:Procedure AddHead(a0:p_List; a1:p_Node);
 -246:Procedure AddTail(a0:p_List; a1:p_Node);
```

```
 -252:Procedure Remove(a1:p_Node);
 -258:Procedure RemHead(a0:p_List);
 -264:Procedure RemTail(a0:p_List);
 -270:Procedure Enqueue(a0:p_List; a1:p_Node);
 -276:Function FindName(a0:p_List; a1:Str):p_Node;
End;
```

Aber keine Panik: Natürlich müssen Sie solche Tabellen nicht ständig selbst schreiben - warum sollten Sie das Rad immer wieder neu erfinden? Die Library-Deklarationen finden Sie in den Includefiles auf einer Ihrer MaxonPASCAL-Disketten. Genaueres dazu im nächsten Kapitel.

Alles, was Sie wissen müssen, ist, daß die Exec- und DOS-Library beim Programmstart immer schon geöffnet sind. Bei den anderen Libraries dürfen Sie nie vergessen, sie am Programmanfang mit „OpenLib" zu öffnen, denn andernfalls ist Ihnen der Guru sicher (ich habe das auch einmal vergessen und bin dann bei der Fehlersuche fast wahnsinnig geworden, weil ich nicht auf die Idee kam, mal am Programmanfang nachzusehen).

8.3 Anmerkungen zu den Include-Files

Die Include-Dateien enthalten alles, was Sie zur Programmierung des AMIGA-Betriebssystems benötigen: Konstanten, Typen (Strukturen) und Libraries. Sie entsprechen im wesentlichen den normalen „C"-Includefiles. Nur hin und wieder mußten die besonderen Gegebenheiten der Sprache PASCAL berücksichtigt werden.

Zuerst wären da Kollisionen mit PASCAL-Schlüsselwörtern zu erwähnen. Die beiden wichtigsten: einige Strukturen (z. B. „NewScreen") enthalten ein Feld mit dem Namen „Type", und in Exec gibt es einen Datentypen namens „Library". Diese Probleme wurden jeweils gelöst, indem diesen Bezeichnern ein Uniterstrich „_" vorangestellt wurde („_Library", „_Type"). Genausq wurden die Problemfälle behandelt, die dadurch entstanden, daß PASCAL nicht zwischen Groß- und Kleinschreibung unterscheidet: in „intuition/intuition.h" gibt es sowohl die Konstante *„CLOSEWINDOW"* als auch die Prozedur *„CloseWindow"* sowie weitere derartige Kollisionen bei *„reportmouse"* und (in „intuition/screens.h") bei *„showtitle"*. In derartigen Fällen hat stets eine Prozedur Vorrang vor einem Datentypen und dieser wiederum vor einer Konstanten, was in den genannten Fällen konkret heißt, daß den Konstantennamen stets ein „_" vorangestellt wurde.

Die dos.library enthält einige Funktionen und Prozeduren, deren Namen mit PASCAL-Bezeichnern kollidieren: *„Read"*, *„Write"*, *„Close"*, *„Seek"*, *„Input"* und *„Output"*. Diese Funktionen erscheinen in den Includefiles jeweils zweimal: zum einen, wie gehabt, mit „_" am Anfang („_Write"), zum anderen mit vorangestelltem „Dos" (z. B. „DosWrite"). Sie haben also die Wahl, welche Namen Sie benutzen wollen.

Weitere Kollisionen: Der Datentyp *„Filehandle"* in der Datei „libraries/dosextens.h" wurde in *„_FileHandle"* umbenannt, und die Funktion *„Text"* der graphics.library kann wahlweise als *„_Text"* oder als *„GfxText"* aufgerufen werden.

Bei MaxonPASCAL gibt es keine Datei „functions.h" wie in C, vielmehr sind die Funktionsdefinitionen auf verschiedene Includefiles verteilt: Die Funktionen der exec.library stehen in verschiedenen Dateien, und zwar jeweils da, wo die zugehörigen Datentypen definiert werden. Sie brauchen sich darüber also normalerweise keine Gedanken zu machen. Für die anderen Libraries wurden entweder „.lib"-Files angelegt, in denen ihre Funktionen deklariert werden, oder sie wurden anderen Includefiles „beigepackt".

Hier ist eine Liste aller zur Verfügung stehenden Libraries, ihrer Basisvariablen und der Dateien, in denen sie deklariert werden:

NAME	BASIS	DATEI
amigaguide.library	AmigaGuideBase	amigaguide.lib
asl.library	AslBase	asl.lib
battclock.library	BattClockBase	battclock.lib
bullet.library	BulletBase	bullet.lib
cardres.library	CardResource	cardres.lib
clist.library	CListBase	clist.lib
colowheel.library	ColorWheelBase	colorwheel.lib
commodities.library	CxBase	commodities.lib
console.library	ConsoleBase	devices/console.h
datatypes.library	DataTypesBase	datatypes.lib
disk.library	DiskBase	disk.lib
diskfont.library	DiskFontBase	libraries/diskfont.h
dos.library	DosBase	libraries/dos.h
dtclass.library	DTClassBase	dtclass.lib
exec.library	SysBase	exec/#?.h
expansion.library	ExpansionBase	libraries/expansionbase.h
gadtools.library	GadToolsBase	gadtools.lib
graphics.library	GfxBase	graphics.lib
icon.library	IconBase	workbench/icon.h
iffparse.library	IFFParse	iffparse.lib
intuition.library	IntBase	intuition.lib
keymap.library	KeymapBase	keymap.lib
layers.library	LayersBase	layers.lib
Locale.library	LocaleBase	locale.lib
mathieeedoubbas.library	MathIeeeDoubBasBase	mathieeedoubbas.lib
mathieeedoubtrans.library	MathIeeeDoubTransBase	mathieeedoubtrans.lib
misc.library	MiscBase	misc.lib
potgo.library	PotgoBase	resources/potgo.h
rexxsyslib.library	RexxSysBase	rexxsyslib.lib
timer.library	TimerBase	devices/timer.h
translator.library	TranslatorBase	libraries/translator.h
utilitiy.library	UtilityBase	utility.lib
workbench.library	WorkbenchBase	wb.lib

Noch eine Bemerkung zu den zahlreichen Datentypen, die in den Includefiles definiert werden: Zu jedem Record-Typen existiert ein Zeigertyp, der den Namen des Record-Typs mit vorangestelltem „p_" trägt, z. B. „Window" und „p_Window". Der Grund dafür ist, daß an einigen Stellen (Parameterlisten) in PASCAL nur Typbezeichner wie z. B. „p_Window", aber keine Typbeschreibungen (wie „^Window") erlaubt sind.

9. Units und Module

9.1 Modulare Programmierung: was, warum und wie?

MaxonPASCAL bietet Ihnen seit Version 2.0 ein Feature, das (wie viele andere nützliche Dinge auch) in Standard-PASCAL fehlt: das Konzept der modularen Programmierung. Die Grundidee ist die, daß ein Programm in beliebig viele sog. Module aufgeteilt wird, die dann mehr oder weniger unabhängig voneinander ediert und compiliert und erst danach zu einem einzigen Programm zusammengebunden werden. Das Zusammensetzen der Module heißt „linken". Sicher ist Ihnen bereits die Ausgabe „Linking..." am Ende des Compilevorgangs von MaxonPASCAL aufgefallen.

MaxonPASCAL ruft den integrierten Linker immer automatisch auf, auch dann, wenn es bei einem nicht-modularen Programm scheinbar nichts zu linken gibt (gibt es aber doch: das Programm wird mit dem PASCAL- Laufzeitsystem zusammengebunden).

Es gibt viele gute Gründe, sein Programm in Module zu zerstückeln:

- Die einzelnen Quelltexte werden kürzer. Da man immer nur das Modul neu compilieren lassen muß, das man gerade bearbeitet, werden auch die Compilezeiten kürzer (bei MaxonPASCAL sowieso kein Thema).

  Auch der Workspace-Speicherbedarf sinkt (bei MaxonPASCAL sehr wohl interessant, denn das Programm benötigt ja einen zusammenhängenden Speicherblock, und da wird es schnell eng).

- Lange Programme werden überschaubarer, wenn man sie sinnvoll in Module aufteilt. „Sinnvoll" heißt hier, daß die Einteilung nicht willkürlich erfolgt, sondern die Module jeweils einen zusammengehörigen Bereich des Programms umfassen. Zum Beispiel könnte man eine Dateiverwaltung in die Module „Benutzeroberfläche", „File-Handhabung" und „Such- und Sortierprozeduren" unterteilen, was sich dann noch beliebig weiter verfeinern und zerkleinern ließe.

- Man kann sich aus immer wieder benötigten Programmteilen eine Bibliothek aus Modulen, die man bei Bedarf nur noch einlinken muß, zusammenstellen. Dafür eignet sich besonders das UNIT-Konzept.

- Viele Leute glauben auch, Module seinen für Programmentwicklungen im Team hilfreich, da dann jeder Programmierer seinen Teil des Projekts unabhängig von den anderen erstellen kann. Nun ja, T.e.a.m. steht bekanntlich für „Toll, ein anderer macht's", und der eine, an dem normalerweise letztendlich die ganze Arbeit hängen bleibt, kann auch ohne Modulkonzept auskommen. Sollten in einem Team jedoch wirklich einmal mehrere Leute arbeiten, so ist Unabhängigkeit alles andere als wünschenswert, denn es bricht regelmäßig ein Chaos herein, sobald die Module zusammengefügt werden sollen (An genau dieser Stelle möchte ich mit einem herzlichen „Core Dumped!" das Juhunix-Propra-Power-Team grüßen). Wenn aber tatsächlich einmal der unwahrscheinliche Fall eintreten sollte, daß ein Team sowohl motiviert

als auch koordiniert arbeitet, könnte sich das modulare Programmieren theoretisch als nutzbringend erweisen.

MaxonPASCAL bietet sogar zwei verschiedene Modulkonzepte, die in gewissen Grenzen miteinander kombiniert werden können. Sie sind gekennzeichnet durch die Bezeichnung UNIT oder MODULE. Zunächst wollen wir uns hier den Modulenn zuwenden, die für den Programmierer zwar etwas schwieriger zu handhaben, aber in gewissem Sinne flexibler einsetzbar sind.

Ein MaxonPASCAL-Programm besteht, da wir vorerst davon ausgehen, daß wir keine Units verwenden, immer aus einem Hauptprogramm und beliebig vielen Modulen. Jeder Teil ist nach dem Compilieren als sog. Objektdatei abzuspeichern, damit der Linker ihn in die anderen Teile einbinden kann. Das Hauptprogramm sieht wie gewohnt aus, während die Module mit „Module" statt „Program" beginnen und keinen Anweisungsteil auf oberster Ebene haben. Die Verbindung der Komponenten geschieht über gemeinsame Variablen und Procedures, die mittels der Directives „IMPORT" und „EXPORT" ausgetauscht werden.

Aber nun eines nach dem anderen!

9.2 Syntax von Modulen

Betrachten wir als erstes ein kleines Beispiel, an dem das Prinzip des Linkens schon klar wird. Hier ist ein Hauptprogramm:

```
Program Haupt;

Var i:integer; EXPORT;

Procedure p(a:integer); IMPORT;
Procedure q; IMPORT;

Function f(n:integer):integer; EXPORT;
   Begin
     f:= n*n
   End;

Begin  { Hauptprogramm }
   p(20);
   q
End.
```

Dieses Hauptprogramm importiert zwei Prozeduren, nämlich „p" und „q", und exportiert die Funktion „f" und die Variable „i". Nun brauchen wir wenigstens noch ein Modul, das die importierten Procedures zur Verfügung stellt (es macht nichts, wenn die exportierten Bezeichner nirgendwo benutzt werden). Hier ist ein solches Modul:

```
Module Unterprogramme;

Var i: integer; IMPORT;
```

```
Procedure p(k:integer); EXPORT;
   Begin
     i:= round(sqrt(k))
   End;

Procedure q; EXPORT;
   Function f(j:integer):integer; IMPORT;  { lokale Funktion! }
   Begin  { von "q" }
     writeln('Funktionswert von ', i, ' ist ', f(i))
   End;

Begin { leerer Anweisungsteil }
End;
```

Zuerst zwei Anmerkungen zur Syntax: Daß die Direktiven „Import" und „Export" hier groß geschrieben wurden, hat (wie immer in PASCAL) absolut keine Bedeutung. Dies geschah nur, damit sie besser auffallen.

Auf oberster Ebene, also da, wo sonst das Hauptprogramm steht, hat das Modul einen leeren Anweisungsteil. Er hätte auch - wie oben erwähnt - ganz weggelassen werden können. Auch haben Sie die Wahl, ob Sie am Ende dieses Pseudo-Hauptprogramms einen Punkt, ein Semikolon (wie oben geschehen) oder auch gar nichts setzen. Es ist nicht möglich, zwischen dieses „Begin End" irgendwelche Befehle zu setzen.

Die Syntax eines Moduls unterscheidet sich davon einmal abgesehen also nicht von der eines Programms:

```
"MODULE" Programmkopf
 Deklarationsteil
 ["BEGIN" [";"] "END" [";"|"."]]
```

Syntax für „Programmkopf":

```
Bezeichner  [ "(" Bezeichner { "," Bezeichner } ")" ]  [ ";" ]
```

Man darf also auch hinter dem Modulnamen (der keine Bedeutung hat) eine Parameterliste wie „(input,output)" setzen - die dann ebenfalls keinerlei Bedeutung hätte.

Eine Prozedur wird importiert, indem man ihren Kopf hinschreibt und anstelle eines Prozedurblocks einfach „Import" schreibt. Als Export wird sie definiert, indem man zwischen ihrem Kopf und Rumpf den Bezeichner „Export" setzt. Ein- und ausgeführte Variablen werden ganz normal deklariert, nur daß hinter die jeweilige Deklarationszeile „Import" bzw. „Export" zu schreiben ist.

In diesem Zusammenhang hat MaxonPascal eine kleine Verbesserung gegenüber Kickpascal 2: Eine Prozedur, die bereits als „Import" deklariert wurde, kann später im Programm noch definiert werden. Sie wird dann exportiert.

Beispiel:

```
  procedure p(i: integer);
 import;
  { ... }
  procedure p; { Parameterliste ist ja schon bekannt}
    begin
      { ... }
    end;
```

Wozu das gut sein soll?

Nun, dadurch können Sie jetzt in Pascal das Modulkonzept von C imitieren. Sie können in eine Includedatei zunächst sämtliche globalen Funktionen als „import" deklarieren und diese Datei dann in jedes Modul includieren. Dadurch sind automatisch alle globalen Funktionen in allen Modulen bekannt und können dann an beliebiger Stelle „für den Export" definiert werden.

Aber zurück zum Beispiel:

Das Hauptprogramm ruft als erstes die Prozedur „p" mit dem Parameter 20 auf. Diese Prozedur wird aus dem Modul importiert und initialisiert die Variable „i", die sie wiederum aus dem Hauptmodul importiert hat. Die Prozedur „q", die das Hauptprogramm als nächstes aufruft, importiert die Funktion „f" lokal, so daß sie nur innerhalb von „q" bekannt ist. Es ist aber nicht möglich, umgekehrt eine lokale Prozedur oder Funktion zu exportieren.

Es sei noch einmal darauf hingewiesen, daß ein Programm stets aus genau (!) einem Hauptprogram, aber beliebig vielen Modulen besteht.

9.3 Objektdateien und Linkersteuerung

Wenn Sie die obigen Listings so von MaxonPASCAL compilieren lassen, ist der Compiler zwar einverstanden, aber der Linker meldet Fehler, weil die importierten Bezeichner nirgendwo definiert werden. Also müssen Sie ihm irgendwie sagen, was er einlinken soll.

Dazu brauchen Sie zunächst einmal Objektdateien, für jeden Bestandteil des Programms einen. Lassen Sie also das „Unterprogramm"-Modul compilieren und ignorieren Sie die Linker-Fehlermeldungen. Jetzt können Sie weder das Programm ausführen noch eine Exe-Datei schreiben lassen. Eine Objektdatei kann man aber unabhängig vom Linker erzeugen lassen. Tippen Sie nach dem Compilieren also „O" (oder wählen Sie das entsprechende Pull-Down-Menü) und speichern Sie die Objektdatei ab. Diese Datei enthält den Programmcode des Moduls und Angaben über die deklarieren Bezeichner.

Laden Sie nun das Hauptprogramm. Jetzt müssen Sie dem Linker sagen, daß die zuvor erzeugte Objektdatei dazu gehört, und zwar mit der Compiler-Anweisung „link":

```
{$link 'pascal:modul.o'}
```

Diese Anweisung können Sie an eine beliebigen Stelle des Quelltextes von „Haupt" setzen. Wenn der Compiler auf diese Directive trifft (siehe dazu auch Kapitel „Compiler-Directives"), lädt er sofort die entsprechende Datei. Es gibt dafür zwei naheliegende Positionen: Entweder nahe der Stelle, wo die zugehörigen „Import"- Deklarationen stehen (ist übersichtlich, weil man dann sofort sieht, was woher kommt) oder als allerletzte Zeile ganz am Programmende. Letzteres ist während der Programmentwicklung praktisch, weil man in der Regel immer wieder neue Compiler-Fehlermeldungen verursacht, während man beim Linken eigentlich nichts mehr falsch machen kann, sobald Im-/Exporte erst einmal stimmen. Deshalb will man so schnell wie möglich wissen, ob das gerade bearbeitete Modul syntaktische Fehler enthält und wird deshalb die Anweisungen zum (bei Diskettenbenutzung erwähnenswerte Zeit erforderndes) Laden von Objektdateien ans Ende setzen, so daß sie erst ausgeführt werden, wenn sonst alles OK ist.

Wie dem auch sei, nun wird der Compiler irgendwann auf die Disk oder Platte zugreifen und die genannte Objektdatei laden. Nach dem Compilieren prüft der Linker wieder, ob alle irgendwo (im Hauptprogramm oder einem eingelinkten Modul) als Import deklarierten Bezeichner irgendwo exportiert werden. Er gibt dann eine vollständige Liste aller undefinierten Bezeichner aus oder, wenn alles klar ist, „Ready". In letzterem Fall haben Sie ein lauffähiges gelinktes Programm vorliegen. Sie können es wie gewohnt starten oder eine Exe-Datei erzeugen lassen.

Die Benutzung von „$link" ist nicht ganz problemlos, wenn Sie Ihre Programmdateien in ein anderes Verzeichnis kopieren und dort neu compilieren wollen. Dann müssen Sie in jedem Quelltext vor dem Übersetzen die Dateinamen in den Linker-Anweisungen von Hand ändern.

Um das zu umgehen, bietet Ihnen „$link" noch eine interessante Option: Wenn der Dateiname der Objectdatei keinerlei Pfadangabe enthält (d. h. im Namensstring weder ein „:" noch ein „/" vorkommt), wird der Pfad des Dateinamens der im Editor gehaltenen Datei vor den Objectdateinamen gehängt (Sätze gibt's...). Beispiel: In einem Quelltext namens „PASCAL:Sources/Chaos/Name.p" taucht die Anweisung {$link „Hilfsmodul.o" } auf. Dann lädt der Linker die Objectdatei „PASCAL:Sources/Chaos/Hilfsmodul.o".

Dem Linker ist es übrigens, im Gegensatz zum Compiler, egal was ein Hauptprogramm und was ein Modul ist. Der Compiler bringt ihm das bei, indem er das Hauptprogramm immer automatisch den Bezeichner „_main" exportieren und jedes Modul diesen Bezeichner importieren läßt. Dadurch ist gewährleistet, daß ein Programm nur korrekt linkbar ist, wenn es einerseits mindestens ein Hauptprogramm (sonst wäre der Bezeichner „_main" ja undefiniert) und andererseits auch höchstens ein Hauptprogramm enthält (sonst würde ein Doppel-Export von „_main" als Fehler gemeldet).

Dieses „_main" markiert übrigens den Anfang des Hauptprogramms, und am Anfang des Codes jedes Moduls steht ein „Jmp" (das Äquivalent im Maschinensprache zum „Goto" in PASCAL) zu eben diesem Symbol. Das hat dann wiederum die Konsequenz, daß Sie in obigen Beispiel auch Hauptprogramm und Modul vertauschen könnten, also erst ein Object-File des Hauptprogramms erzeugen und es dann in das Modul einlinken lassen.

In der praktischen Programmierung wird es sinnvollerweise so aussehen, daß Sie Objektdateien von allen Modulen und vom Hauptprogramm anlegen und dann den Quelltext mal der einen, mal der anderen Komponente im Editor bearbeiten. Dabei enthält jeder einzelne Quelltext die Compiler-Directives, die alle anderen Teile einladen. Wenn Sie jeweils einen Text verbessert und ausgetestet haben und an einem anderen Programmteil weiterarbeiten wollen, sollten Sie vor dem Laden des nächsten Quelltexts zuerst das Ergebnis Ihrer Arbeit als Objektdatei abspeichern. Das geht vom Tastaturmenü ja sehr bequem, wenn Sie sich an den vorgeschlagenen Dateinamen halten: Es kostet Sie nur die beiden Tastendrücke „O" und „Return" - und natürlich, falls noch nicht geschehen, auch noch die Taste „S" zum Abspeichern der neuesten Version des Quelltextes.

Eine Objektdatei enthält immer nur den Code des zuvor compilierten Programmteils, nicht etwa auch die mit {$link } eingeladenen Codestücke oder das MaxonPASCAL-Laufzeitsystem. Sie brauchen also beim oben beschriebenen Programmierablauf nicht zu befürchten, daß ein Teil im Code doppelt auftaucht.

Seit MaxonPascal 3 ist „$link" aber ziemlich überflüssig, denn es gibt ja die Projektverwaltung „Make". „$link" wird im Make-Modus ignoriert.

9.4 Alink-Kompatiblität - Fluch und Segen

Der integrierte Linker von MaxonPASCAL ist im wesentlichen zum AMIGA-Standardlinker „Alink" bzw. der PD-Version „Blink" kompatibel, wenn er auch einige weniger wichtige Features dieses Linker-Standards nicht unterstützt. Das hat die erfreuliche Konsequenz, daß Sie problemlos Assembler-Routinen einlinken können (siehe übernächstes Kapitel), denn fast alle AMIGA-Assembler erzeugen Alink-kompatible Objektdateien. Aber es gibt auch einen Nachteil: Da der Alink im Prinzip für alle Programmiersprachen verwendet wird, hat er keinerlei Vorstellung davon, was eine Prozedur oder eine Parameterliste ist.

Seine Arbeit beschränkt sich darauf, Bezeichner zuzuordnen: Wenn ein Bezeichner irgendwo benutzt und anderswo definiert wird, trägt er an der ersten Stelle die entsprechende Adresse ein - sonst nichts.

Das bedeutet für Sie als PASCAL-Programmierer, daß Sie selbst auf die modulüberschreitende Semantik achten müssen. Sie könnten zum Beispiel versehentlich in einem Modul eine Variable exportieren und in einem anderen Modul eine gleichnamige Prozedur importieren. Ergebnis: Der Compiler weiß nichts über die anderen Module und meldet deshalb keinen Fehler, der Linker, der keine Ahnung von PASCAL hat, stellt hier eine Namensgleichheit fest. Zur Laufzeit wird das Programm dann mit einen Unterprogrammaufruf in einen Datenbereich springen - ein Fall für den Exception-Handler.

Ein anderes Beispiel: ein Modul exportiert eine INTEGER-Variable (mit 2 Bytes Speicherbedarf), ein anderes importiert diese als 4 Byte lange LONGINT-Variable. Wenn das importierende Modul nun lesend auf die Variable zugreift, ist das Ergebnis undefiniert, und wenn es schreibend zugreift (Wertzuweisung), wird über den für die INTEGER- Variable reservierten Bereich hinaus geschrieben - mit unabsehbaren Folgen.

Auch ist es wichtig, daß beim Im- und Export von Prozeduren und Funktionen die Parameterlisten der Import- und Export-Deklaration äquivalent sind - auch hier können Sie sonst „nette" Überaschungen erleben. Aus all' dem folgt für Sie, daß Sie bei Im- und Exporten aller Art höllisch aufpassen müssen.

Als PASCAL-Programmierer sind Sie es gewohnt, daß „Test", „test" und „TEST" lediglich verschiedene Schreibweisen für den selben Bezeichner sind, denn in PASCAL ist die Groß-/Kleinschreibung ja egal. Nun wurde der Alink ursprünglich für C und Assembler geschrieben, und da ist (bei C immer, bei den AMIGA-Assembler meistens) die Schreibweise eines Bezeichners signifikant. Folglich mag der Alink es nicht, wenn man eine Prozedur „Hallo" exportiert, aber „hallo" importiert - für ihn sind dies völlig verschiedene Bezeichner.

Das ist natürlich nicht schön. Deshalb hat der ins MaxonPASCAL-System integrierte Linker eine (Default-mäßig eingeschaltete) Option, die ihn veranlaßt, die Groß-/Kleinschreibung zu ignorieren. Diese Option wird mit dem Pull-Down-Untermenü „Optionen/Linker" geschaltet. Wenn man sie ausschaltet („GROSS/klein beachten" angewählt), müssen die Bezeichner in der IMPORT-Zeile genau so wie in der EXPORT-Deklaration geschrieben werden - wie man ihn im Rest der Module schreibt, ist Sache des Compilers und somit dem Linker egal.

Vielleicht fragen Sie sich jetzt, warum es diese gegen alle lieben Gewohnheiten des PASCAL-Programmierers verstoßende Option überhaupt gibt. Well, vielleicht haben Sie einmal das Bedürfnis, Ihr Programm extern zu linken, z. B. im Zusammenhang mit der Einbindung von Assembler-Routinen. Dann will man keine bösen Überraschungen erleben, sondern schon in der MaxonPASCAL-Entwicklungsumgebung erfahren, ob der externe Linker später Fehler melden wird - also sollte sich der MaxonPASCAL-Linker dann genauso verhalten wie sein Kollege Alink.

9.5 Hunks

Dieser Abschnitt ist für den „normalen" PASCAL-Programmierer nicht unbedingt nötig. Ich werde hier die Einträge erklären, die plötzlich in der Speicherbelegungsliste auftauchen, wenn mit Units oder Modulen gearbeitet wird.

AMIGA-Programmdateien - also z. B. die von MaxonPASCAL erzeugten Exe-Dateien - sind in sogenannten „Hunks" organisiert. Es gibt im wesentlichen drei Hunktypen:

- CODE: Ein solcher Hunk enthält lauffähigen Programmcode.
- DATA: Hier können die Daten eines Programms liegen. In Maschinencode gibt es aber keinen prinzipiellen Unterschied zwischen Programm und Daten, so daß dieser Hunktyp eigentlich mit „CODE" identisch ist.
- BSS: Auch hier liegen Daten, aber solche, die beim Programmstart keinen definierten Inhalt haben. Ein BSS-Hunk ist deshalb eigentlich nicht mehr als eine Anweisung an die Laderoutine, Speicher zu reservieren, in dem das Programm dann z. B. Variablen unterbringen kann.

Für Leute, die gern in Hexdumps wühlen: Ein „CODE"-Hunk beginnt in der Datei mit dem Langwort $000003E9, „DATA" mit $000003EA und die Kennung $000003EB steht für „BSS". Das Langwort nach der Hunk-Kennung enthält jeweils die Hunklänge in Langworten, danach kommt der Inhalt des Hunks (außer natürlich bei „BSS", der ja keinen definierten Inhalt hat). Hinter jedem Hunk kann noch eine Relokationstabelle (Kennung: $000003EC) stehen, die Angaben dazu enthält, wie der Hunkinhalt zu modifizieren ist, um ihn an eine Speicheradresse anzupassen - denn im AMIGA wird ja grundsätzlich nicht mit festen Speicheradressen gearbeitet - und wie die Hunks verbunden sind. Bitte verwechseln Sie „Hunk" nicht mit „Hrung", zumal sowieso keiner weiß, was ein Hrung ist und warum er gerade auf Beteigeuze 7 explodiert ist.

Was hat das Ganze mit MaxonPASCAL und Modulen zu tun? Nun, auch Objektdateien sind aus Hunks aufgebaut, nur daß die Hunks zum Teil nicht über Relokationstabellen, sondern über Verweise auf Bezeichner verbunden sind, so daß der Linker im wesentlichen die Aufgabe hat, gleiche Symbole zu verbinden und daraus Relokationstabellen zu erstellen.

9.6 Dynamische und statische Variablen

Normalerweise verwalten MaxonPASCAL-Programme ihre Variablen auf dem Stack. Beim Programmstart wird dort genug Platz für die globalen Variablen eingerichtet, und immer, wenn eine Prozedur oder Funktion aufgerufen wird, reserviert sie sich als erstes die benötigte Menge Speicher. Durch dieses Variablenkonzept ist es möglich, daß jede Schachtel einer rekursiven Prozedur ihre eigenen Variablen hat.

Der Compiler berechnet, wieviel Stack das Programm ohne Rekursion maximal (d. h. zum Zeitpunkt der maximalen Schachtelungstiefe) braucht, und gibt diesen Speicherbedarf als „Linear stack requirement" aus.

Daneben gibt es noch ein weiteres Variablenverwaltungskonzept: Statische Variablen. Solche Variablen liegen in einem BSS-Hunk, sind also Bestandteil der Exe- oder Objektdatei und werden somit schon beim Laden des Programms eingerichtet. Jede statische Variable existiert dadurch immer genau einmal, auch lokale Variable einer rekursiven Prozedur.

Exportierte Variable sind immer statisch, denn der Linker muß ja schon nach dem Compilen wissen, wo sie zur Laufzeit im Speicher liegen, was bei dynamisch auf den Stack gelegten Variablen natürlich nicht möglich wäre. Ferner sind die „globalen" Variablen in Modulen (also die Variablen, die dort außerhalb einer Prozedur oder Funktion deklariert werden), stets statisch. Grund: Um sie auf dem Stack zu verwalten, müßte jedes Modul zur Compilezeit „wissen", wie viel Platz alle anderen Module für ihre Variablen brauchen und wo noch Platz für die eigenen frei ist.

Von diesen inhärent statischen Variablen merken Sie in der Regel nichts, außer, daß die Speicherbelegungsliste einen entsprechend großen BSS-Hunk ausweist oder daß Sie Schwierigkeiten bekommen, falls Sie auf die Idee kommen sollten, eine lokale Variable einer rekursiven Prozedur zu exportieren. Sie können Variablen aber auch bewußt als statisch deklarieren: mit der Directive „STATIC". Sie wird genau wie „Import" oder „Export" hinter die Variablendeklaration gesetzt, z. B. so:

```
Program Statisch;

Var v: integer;

Procedure p(flag: Boolean);
   Var v: integer; STATIC;
   Begin
     If flag Then v:= 26731
             Else writeln(v)
   End;

Begin
   v:= 42;
   p (true);
   p (false);
   writeln (v)
End.
```

Dieses Programm erzeugt die Ausgabe:

```
26731 42
```

Wie funktioniert es? Die Prozedur „p" hat eine lokale Variable „v", die nichts mit der gleichnamigen globalen Variablen zu tun hat. Beim ersten Aufruf mit dem Parameter „true" wird diese lokale Variable mit dem Wert 26731 initialisiert. Dadurch aber, daß „v" statisch ist, existiert es nach dem Ende der Prozedur weiter und hat beim zweiten Aufruf immer noch den selben Wert - andernfalls wäre er undefiniert.

Die globale Variable „v" habe ich im Beispiel eingeführt, um zu zeigen, daß das statische „v" zwar zur gesamten Laufzeit existiert, aber trotzdem nicht global ist.

Übrigens: Wenn Sie das „STATIC" wegließen, würde sich in diesem Fall nichts an der Ausgabe ändern, denn zur dynamischen Erzeugung des lokalen „v" wird immer derselbe Speicherbereich benutzt. Zwischen den beiden Aufrufen von „p" wäre dieser Speicher aber ungeschützt und akut überschreibgefährdet.

9.7 Externes und internes Linken mit „PasLib.o"

Auf Ihrer MaxonPASCAL-Disk finden Sie eine Datei „paslib.o". Sie enthält das gesamte Laufzeitsystem von MaxonPASCAL, also eine Sammlung von Unterprogrammen wie z. B. Ein- und Ausgabeoperationen aller Art, die dem PASCAL-Programmierer zur Verfügung stehen, der Prozessor aber nicht ohne weiteres selbst kann. Beispielsweise ist eine Addition zweier Integer-Zahlen als ein einziger Maschinensprachbefehl zu realisieren, während die Ausgabe einer solchen Zahl eine sehr komplexe Operation und nur als ein längeres Unterprogramm realisierbar ist.

Ein identisches Laufzeitsystem ist integrierter Bestandteil des MaxonPASCAL-Systems, so daß der Linker es ohne Diskettenzugriff mit dem vom Compiler erzeugten Code und den evtl. zugeladenen Modulen zusammenbinden kann. Die Datei „paslib.o" brauchen Sie deshalb nur, wenn Sie einen

anderen als den integrierten Linker benutzen, z. B. um mit dem MaxonASS eine Assembler-Unterroutine zu schreiben und aus dem MaxonASS-System heraus zu debuggen (mehr dazu im folgenden Abschnitt).

Sie können auch „einfach mal so" extern linken: besorgen Sie sich „BLink", erzeugen Sie zu einem PASCAL-Programm eine Objektdatei, z. B. „hello.o", und tippen Sie im CLI:

```
BLink hello.o, paslib.o to Hello
```

Auf diese Weise erhalten Sie aus der Objekt-Datei eine lauffähige Exe-Datei. Nun benötigt wohl kein einzelnes Programm sämtliche Möglichkeiten, die MaxonPASCAL bietet. So wird man in der Regel entweder „REAL" oder „LONGREAL" benutzen, nicht aber beide Zahlenformate durcheinander - falls man überhaupt Fließkomma-Operationen braucht. Um in Exe-Files nicht zu viel unnützen Code stehen zu haben, ist das ca. 10 KByte große MaxonPASCAL-Laufzeitsystem in mehrere „Bibliotheken" unterteilt: Zum einen wäre da die knapp 4 KByte umfassende Grundlibrary. Somit gibt der Linker immer eine „Maximum Code Size" von wenigstens 3900 Byte aus. Diese Bibliothek enthält im wesentlichen die wichtigsten Ein-/Ausgabeoperationen, die Initialisierungs- und die Endprozedur, welche am Anfang bzw. Ende des Programms aufgerufen werden, die meisten Laufzeit-Fehlermeldungen und so fort. Sie exportiert zum Linker das Symbol „_paslibbase", das ihre Basisadresse enthält.

Oberhalb dieser Adresse, die am Programmanfang ins Prozessorregister A5 geladen wird und dort unverändert stehen bleiben muß, stehen verschiedene wichtige Daten wie z. B. der „Global Vector", über den MaxonPASCAL die Stack-Variablen verwaltet; unterhalb der Basis steht eine Sprungtabelle zu den einzelnen Funktionen.

Außerdem gibt es noch sechs weitere, nach „Themen" geordnete Bibliotheken, die ebenfalls jeweils ihre Basisadresse exportieren:

_reallibbase:	Fließkommaoperationen, einfache Genauigkeit
_doublelibbase:	Fließkommaoperationen, doppelte Genauigkeit
_setlibbase:	alle SET-Operationen
_stringlibbase:	größere Stringoperationen wie „+", „Insert" usw.
_misclibbase:	diverses, z. B. „Random", „Hi", „ClrEoL", ...
_amigalibbase:	AMIGA-spezifisches (Windows, Messages, Console...)

Welche Konsequenzen hat das für Sie als PASCAL-Programmierer? Zum einen dürfen Sie die oben genannten Bezeichner (und „_main") nicht aus PASCAL-Modulen exportieren, denn sonst kommt es zu Konflikten.

Wichtiger ist aber, daß der interne Linker nur die Bibliotheken des Laufzeitsystems einbindet, aus denen auch Funktionen benutzt werden, während man bei externem Linken nur die Möglichkeit hat, alles einzubinden.

9.8 Units

Das Unit-Konzept ist eine sehr leistungsfähige Form der modularen Programmierung. Hier werden Compiler und Linker durch spezielle Sprachkonstrukte näher aneinander gebunden, wodurch viele Fehlermöglichkeiten, die es bei den MODULEn gibt, ausgeschlossen werden. Außerdem bieten Units eine bequeme Möglichkeit, den Sprachumfang von MaxonPASCAL durch Anlagen von Unit-Bibliotheken nahezu beliebig zu erweitern.

9.8.1 Ein erstes Beispiel

Analog zu den MODULEn besteht ein MaxonPASCAL-Programm aus einem Hauptprogramm und beliebig vielen Units. Hier ist ein erstes Beispiel für ein Unit:

```
UNIT Beispiel1;

 { Ein Unit beginnt mit dem Schlüsselwort "UNIT", gefolgt vom
   Unitnamen. Während Programm- und Modulnamen bedeutungslos sind,
   werden die Unitnamen für mehrere Zwecke benötigt, so daß man
   sie sinnvoll wählen sollte. }

INTERFACE

  { Ein Unit besteht im wesentlichen aus zwei Teilen: dem
    "öffentlichen" (also dem Hauptprogramm und anderen Units
    zugänglichen) Interface-Teil, und dem "privaten" (für andere
    Programmteile verborgenen) Implementation-Teil. }

 TYPE Natuerlich = 0..MaxInt;
  { Dieser Datentyp wird im Interface-Teil des Units "Beispiel1"
    definiert und steht damit allen Programmen, die dieses Unit
    benutzen, zur Verfügung. }

 VAR Grenze: Natuerlich;
  { Dies ist eine Variable des Units "Beispiel1" }

 PROCEDURE Eingabe (VAR Zahl: integer);
  { Diese Prozedur wird vom Unit "Beispiel1" exportiert. Im
    Interface-Teil steht nur ihr Prozedurkopf, der "Rumpf"
    folgt im Implementationsteil. }

 FUNCTION IstPrim (n: Natuerlich): Boolean;
  { Auch diese Prozedur wird von diesem Unit exportiert. }

IMPLEMENTATION

  { Hier beginnt der zweite, nicht-öffentliche Teil des Units. Er
    enthält die Rümpfe der im Interfaceteil deklarierten Prozeduren
    und Funktionen sowie alle Bezeichner dieses Units, die nicht
    exportiert werden sollen. }

 FUNCTION Teilbar (a, b: Natuerlich): Boolean;
```

```
    { Diese Funktion gibt an, ob a durch b teilbar ist. }
    BEGIN
       Teilbar:= (a MOD b = 0)
    END;

  FUNCTION IstPrim;
    { Jetzt kommt der Rumpf dieser im Interfaceteil deklarierten
      Funktion. Beachten Sie, daß der Funktionskopf hier keinerlei
      Parameter und keinen Ergebnistypen enthält, denn das wurde
      alles ja schon im Interfaceteil definiert.
      Die Funktion soll ermitteln, ob die natürliche Zahl "n" eine
      Primzahl ist. }
    VAR Zaehler: integer;
    BEGIN
      Zaehler:= 2;
      WHILE (Zaehler < n DIV 2) AND NOT Teilbar(n, Zaehler) DO
      Zaehler:= Zaehler+1;
      IstPrim:= NOT Teilbar(n, Zaehler)
    END;

  PROCEDURE Eingabe;
    { Auch hier wird die Parameterliste nicht noch einmal geschrieben }
      BEGIN
      Write('Bitte eine Zahl eingeben: ');
      Readln (Zahl)
    END;

END. { Dies ist das Ende des Units. }
```

Wenn der Compiler dieses Unit fehlerfrei übersetzt hat (hier können Sie es ignorieren, wenn der Linker Fehler meldet, denn ein Unit kann man sowieso nicht direkt starten), legt er automatisch zwei Dateien an: Die Objectdatei „Beispiel1.o", die wie bei Modulen den von Compiler erzeugten Code des Units enthält, und die Interface-Datei „Beispiel1.u". Darin steht in vorcompilierter Form eine Kopie des Interface-Teils des Units.

Aber wohin werden diese Dateien geschrieben? Im Pull-Down-Menü „Optionen" finden Sie den Menüpunkt „Suchpfade". Wenn Sie diesen mit der Maus anwählen, erscheint ein kleines Fenster mit je zwei Textgadgets für Includefiles und für Unit-Dateien. Hier interessieren uns nur die beiden Zeilen unter „Unit".

Falls Sie nur die erste Zeile benutzen, ist alles einleuchtend: Dann werden die beiden oben genannten Dateien in dieses Verzeichnis geschrieben. Wenn Sie zwei Pfade angeben, wird wieder derselbe raffinierte Trick wie bei den Include-Files angewendet: Die Dateien werden doppelt erzeugt, und zwar in den beiden angegebenen Verzeichnissen. Nun werden die Dateien ja nicht „einfach so" erzeugt, sondern sollen auch irgend wann einmal gelesen werden (dazu später mehr). Zuerst sucht das PASCAL-System im ersten und dann im zweiten angegebenen Verzeichnis nach den Dateien. Wenn sie nur im zweiten Verzeichnis gefunden werden, werden sie ins erste umkopiert (nicht umgekehrt!). Also können Sie als ersten Pfad ein Unterverzeichnis in der schnellen RAM-Disk und als zweiten ein Directory auf Ihrer Festplatte bzw. PASCAL-Arbeitsdiskette wählen. Bequemer ist es

natürlich, wenn Sie die Pfade nicht nach jedem Start des MaxonPASCAL-Systems neu eintragen, sondern mit dem Programm „PASCALPrefs" die richtige Einstellung in Ihr Config-File eintragen.

Übrigens werden bei der Unit-Handhabung wie beim Laden von Includefiles Unterverzeichnisse bei Bedarf vom PASCALsystem automatisch erzeugt.

9.8.2 USES

Nun haben wir also ein Unit in Form von zwei bzw. vier Dateien vorliegen. Jetzt können wir es ganz einfach in einem unserer Programme benutzen, z. B. so:

```
PROGRAM Primzahl;

USES Beispiel1;

VAR i: Natuerlich;

BEGIN
   Eingabe (i);
   IF IstPrim (i) THEN
     Writeln (i, ' ist prim.')
   ELSE
     Writeln ('Keine Primzahl.')
 END.
```

Sobald der Compiler auf die „USES"-Anweisung trifft, liest er die Datei „Beispiel1.u" wie ein Include-File ein. Dadurch stehen im folgenden die Bezeichner „Natuerlich", „Grenze" (wird hier nicht benutzt), „Eingabe" und „IstPrim" zur Verfügung. Dabei werden Variablen, Prozeduren und Funktionen automatisch als „Import" behandelt.

Außerdem veranlaßt die USES-Anweisung den Compiler, zum Schluß noch die Datei „Beispiel.o" einzuladen.

9.9 Units - Alles noch 'mal etwas genauer

9.9.1 Syntax

Ein Unit beginnt immer mit dem Wort „UNIT", gefolgt vom Unit-Namen, der ein Bezeichner ist. Dahinter beginnt automatisch der Interface-Teil. Sie können dies dadurch deutlich machen, daß Sie das Wort „Interface" schreiben. Dies kann aber auch ersatzlos entfallen.

Danach können USES-Zeilen folgen, denn ein Unit kann selbst wieder andere Units benutzen. Diese Zeilen haben folgende Syntax:

```
Uses-Deklaration= { [ "FROM" Bezeichner ] "USES" Bezeichner { "," 
Bezeichner } ";" }
```

Ein Beispiel:

```
USES Intuition, Graphics; USES Beispiel1;
```

Was es mit dem „FROM" auf sich hat, werde ich später verraten.

Nun folgt der Rest des Interface-Parts. Hier kann folgendes stehen:

- Konstanten
- Datentypen
- Variablen
- Librarys
- Prozedur- und Funktionsköpfe

Wie üblich sind Sie an keine feste Reihenfolge gebunden. Labels können hier nicht deklariert werden.

Also hat der Interfaceteil eines Units diese Syntax:

```
Interfaceteil =
  [ "INTERFACE"]  Uses-Deklaration  { Konstantendefinition |
  Typdefinition | Variablendeklaration | Librarydeklaration |
  Prozedurkopf | Funktionskopf }
```

Intern behandelt der Compiler die hier deklarierten Funktionen und Prozeduren wie FORWARD-Deklarationen. Sie können aber auch als IMPORT bzw. EXTERNAL deklariert werden, aber nicht als EXPORT, denn sie werden ja sowieso exportiert.

Mit dem Wort „IMPLEMENTATION" beginnt dann der andere Teil des Units. Er ist wie ein normaler Deklarationsteil eines Blocks aufgebaut und endet entweder mit einen „END" oder dem Initialisierungsteil (dazu später mehr). Als letztes darf (!) ein Punkt oder ein Semikolon stehen:

```
Implementationsteil =
 "IMPLEMENTATION" Deklarationsteil ("END" | Anweisungsteil) ["." |
";"]
```

Damit hätten wir auch schon die vollständige Syntax für ein Unit:

```
Unit = "UNIT" Bezeichner [ ";" ] Interfaceteil Implementationsteil
```

Im Implementationsteil müssen alle Prozeduren und Funktionen, deren Kopf im Interfaceteil deklariert wurde, definiert werden. Die Reihenfolge der einzelnen Prozeduren ist dabei egal. Hier besteht ihr Kopf aber nur noch aus dem Schlüsselwort PROCEDURE bzw. FUNCTION und dem Namen, die Parameterliste muß (und darf) nicht noch einmal angegeben werden.

9.9.2 Units und Module

Units können von Programmen, Modulen und anderen Units mittels USES aufgerufen werden. Zwei oder mehr Units dürfen sich aber nicht gegenseitig aufrufen. Es besteht also teilweise eine Art Hierarchie unter den Units eines Programms, während MODULEs als gleichrangig behandelt wurden.

Ein Unit-Aufruf aus einem MODULE könnte so aussehen:

```
Module Modul;

Uses Crt;

Procedure Ausgabe; Export;
   Begin
     writeln('Hallo!');
     delay(50)
   End;

Begin End      { Ende des Moduls }
```

Dieses Modul benutzt also das Standard-Unit „Crt", das besondere Prozeduren und Funktionen für die Bildschirm-Ein-/Ausgabe definiert. Ein geeignetes Hauptprogramm zu diesem Modul könnte wie folgt aussehen:

```
Program Haupt;

Uses Crt;

Procedure Ausgabe; Import;

Begin
   Ausgabe
End.

{$link "Modul.o" }
```

Obwohl das Hauptprogramm das Unit „Crt" nicht direkt benutzt, muß in diesem Fall die entsprechende USES-Anweisung gesetzt werden. Dadurch weiß der Compiler, daß er den Code dieses Units laden und dessen Initialisierungsteil (dazu später mehr) aufrufen muß.

Wenn Sie die neue Projektverwaltung „Make" verwenden, müssen Sie alle Units, die irgendwo im Projekt benutzt werden, in das Gadget „Make-Units" des Compilerrequesters eintragen, damit der Linker weiß, was er laden muß.

9.9.3 Schachtelung von USES-Aufrufen

Nun wollen wir uns noch ansehen, was genau der Compiler beim Aufruf von Units macht. Dazu betrachten wir einige Sourcefragmente:

```
Unit A;
Interface
{ Definitionen von Unit A }
Implementation
End.

Unit B;
Interface
Uses A;
{ Definitionen von Unit B }
Implementation
End.

Unit C;
Interface
Uses B;
{ Definitionen von Unit C }
Implementation
 End.

Unit D;
Interface
Uses C,A;
{ Definitionen von Unit D }
Implementation
End.

Program Hauptprogramm;
Uses B,D;
Begin { Hauptprogramm }
End.
```

Als erstes müssen wir hier das Unit A compilieren, da alle anderen Programmteile direkt oder indirekt davon Gebrauch machen. Das Unit B benutzt nur Unit A und kann also direkt danach übersetzt werden.

Erst jetzt kann Unit C compiliert werden. Dabei geschieht folgendes: Durch „Uses B" wird der Compiler angewiesen, den Interface-Teil von Unit B zu übersetzen. Dort steht als erstes „Uses A", so daß auch noch der Interfaceteil von Unit A und erst dann der Rest vom Interface des Units B gelesen wird. Dann wird das Unit C zu Ende übersetzt, wobei dem Compiler die Interface-Deklarationen der Units A und B bekannt sind.

Interessant wird es nun beim Unit D, das erst C und dann A mittels „Uses" benutzt. Zuerst wird Unit C aufgerufen, welches Unit B, das wiederum Unit A benutzt, voraussetzt. Es werden durch die Anweisung „Uses C" also nacheinander die Interfaceteile von A, B und C (in dieser Reihenfolge)

gelesen. Nun geht es weiter im Unit D: es soll zusätzlich noch Unit A benutzt werden. Nun merkt der Compiler aber, daß A schon gelesen wurde und liest es deshalb nicht noch einmal. Somit ist „Uses A" hier überflüssig. (Es hat aber doch eine Wirkung: Das Unit A wird an die erste Stelle der Unit-Liste gesetzt, was wichtig sein kann, wenn mehrere Units gleichlautende Bezeichner definieren. Dazu im nächsten Abschnitt mehr.)

Zuletzt betrachten wir noch das Hauptprogramm. Es benutzt die Units B und D und damit - wie wir bereits wissen - indirekt auch die beiden anderen. Zuerst liest es also das Unit A, um B lesen zu können. Bevor danach D ge-used wird, muß nur noch C gelesen werden, denn A und B sind ja bereits bekannt. Die Reihenfolge der benutzten Units ist also A-B-C-D, genau die selbe, als wenn man im Hauptprogramm nur „Uses D" geschrieben hätte. Somit ist die „Uses B"-Anweisung hier völlig überflüssig.

9.10 Bezeichner-Handhabung in Units

Units sind wunderbar dazu geeignet, den Befehlsumfang von MaxonPASCAL den eigenen Bedürfnissen entsprechend zu erweitern: Man schreibe sich ein Unit mit den neuen Prozeduren, Funktionen, Datentypen etc. und übersetze es einmal - schon kann man es in jedes Programm mit einer einzigen USES-Anweisung einbinden und hat die exportierten Bezeichner dieses Units zur Verfügung. Man kann sogar Units an andere Programmierer weitergeben (sogar verkaufen...), ohne die Quelltexte herausgeben zu müssen: es genügt, wenn der Benutzer die Dateien „MeinUnit.u" und „MeinUnit.o" bekommt.

Stellen Sie sich aber einmal folgendes vor: Von einem Freund haben Sie ein Unit „Matrix" bekommen, das Prozeduren für Matrixoperationen enthält. Auf einer PD-Diskette fanden Sie ein Unit „LongMath", das Prozeduren für das Rechnen mit beliebig großen natürlichen Zahlen definiert. Da Sie ein Mathematik-Freak sind, wollen Sie nun in einem Programm beide Units benutzen. Beim Durchlesen der Dokumentationen zu den beiden Units erschrecken Sie: beide Units definieren eine Prozedur oder Funktion namens „Mult", nämlich für die Multiplikation von Matrizen bzw. langen Zahlen. Was sollen Sie jetzt tun, zumal Sie von keinem der Units den Quelltext haben, die Namen der Prozeduren also nicht ändern können?

Ich kann Sie beruhigen: das Ganze ist kein Problem, denn Units dürfen ohne weiteres identische Bezeichner exportieren. Sie können die beiden oben genannten Units also etwa so in Ihr Programm einbinden:

```
PROGRAM Mein_geniales_Mathematikprogramm;

USES Matrix,     { Geschrieben von Kasimir Müller, Juli 1990    }
  LongMath;  { Gefunden auf der PD-Disk "Quickstart #0815" }
```

Nun bezieht sich der Bezeichner „Mult" auf das als letztes geladene Unit, also „LongMath". Das ist natürlich noch immer nicht schön, denn im Zweifelsfall wollen Sie in Ihrem Programm sowohl Matrizen als auch große Zahlen multiplizieren. Dafür gibt es in MaxonPASCAL ein besonderes Sprachkonstrukt: Qualifizierte Bezeichner.

Ein Bezeichner wird „qualifiziert", indem man ihn den Namen des Units, in dem er definiert wird, voranstellt - mit einem Punkt getrennt. Also können Sie in „Ihrem genialen Mathematikprogramm" mit „Matrix.Mult" und „LongMath.Mult" gezielt auf die beiden Prozeduren zugreifen.

Globale Bezeichner von Programmen können durch Voranstellen des Programmnamens „qualifiziert" werden, und die vordefinierten Standardbezeichner von MaxonPASCAL bilden ein Pseudo-Unit namens „System". Also kann man durch selbstquälerische Qualifizierungen sogar Standardbezeichner überdefinieren und trotzdem benutzen, wie es im folgenden Beispiel mit der Prozedur „Writeln" geschieht:

```
PROGRAM Qual;

VAR Writeln: integer;

BEGIN
   Qual.Writeln:= 42;
   System.Writeln(Qual.Writeln)
END.
```

Es dürfte problemlos konsensfähig sein, daß qualifizierte Bezeichner nicht gerade das Gelbe vom Ei sind, zumal sie zusätzlichen Schreibaufwand erfordern, was Freaks sowieso vermeiden, wo sie nur können. Deshalb sollten sie eine Notlösung in Fällen wie dem oben beschriebenen sein, während man bei selbstgeschriebenen Units darauf achten sollte, daß die exportierten Bezeichner nicht mit denen anderer Units kollidieren.

Für Leute, die alles ganz genau wissen wollen, soll jetzt noch erläutert werden, wie der Linker mit Units umgeht. Bekanntlich legt der Compiler für jedes Unit zwei Dateien an: Die Interface-Datei mit der Endung „.u" und die Objektdatei, zu erkennen an der Endung „.o".

Erstere wird von Compiler gelesen, sobald er auf eine entsprechende USES-Anweisung stößt, letztere wird anschließend vom Linker in das Hauptprogramm eingebunden. Dabei werden wie bei den Modulen Bezeichner an den Linker übergeben, und zwar alle Namen der Variablen, Prozeduren und Funktionen des Interfaceteils - die dort definierten Konstanten, Datentypen und Librarys sind ja nur für den Compiler von Interesse.

Wenn Sie die Kapitel über Module aufmerksam gelesen haben, wissen Sie vielleicht noch, daß der Linker doppelte Deklarationen gleichnamiger Bezeichner als Fehler meldet. Dagegen können, wie oben beschrieben, verschiedene Units durchaus denselben Bezeichner exportieren. Um dieses Problem zu umgehen, werden die Bezeichner „qualifiziert" an den Linker übergeben. Im obigen Beispiel (dem „genialen" Matheprogramm) heißt das, daß an den Linker die beiden Bezeichner „Matrix.Mult" und „LongMath.Mult" übergeben werden.

Bekanntlich können Sie über ein Pull-Down-Menü einstellen, ob der Linker die Groß-/Kleinschreibung von Bezeichnern beachten soll. Falls Sie nun den weniger toleranten Modus wählen, können Sie eine Überraschung erleben, wenn Sie in der Uses-Zeile die Unitnamen falsch schreiben, z. B. „USES matrix": Der Compiler erfährt dann zwar aus dem Interfacefile, daß die zu importierende Prozedur „Mult" heißt, glaubt aber fälschlich, das Unit heiße „matrix". Also wird ver-

sucht, in das Hauptprogramm den Bezeichner „matrix.Mult" zu importieren - exportiert wird aber lediglich der Name „Matrix.Mult". Also meldet der Linker einen undefinierten Bezeichner.

Aus demselben Grunde ist es auch nicht möglich, einem compiliert vorliegenden Unit einen anderen Namen zu geben: Sie können zwar auf der Diskette die Dateien umbenennen, etwa von „Matrix.*" in „Mat.*", und diese Units dann mit „USES Mat" benutzen. Die in der Datei „Matrix.o" exportierten Bezeichner tragen aber weiterhin Namen wie „Matrix.xxx", während Ihr Programm nun „Mat.xxx" importieren will. Ergo meldet der Linker massenhaft Fehler.

9.11 Initialisierung und andere Spezialitäten

9.11.1 Initialisierungs-Prozeduren

In der formalen Syntax von Units wurde bereits angedeutet, daß man Units mit so etwas wie einem Hauptprogramm versehen kann.

Beispiel:

```
UNIT Beispiel2;

INTERFACE

VAR n: integer;

PROCEDURE Ausgabe;

IMPLEMENTATION

PROCEDURE Ausgabe;
   BEGIN
     Writeln ('n = ', n)
   END;

BEGIN    { Initialisierungsteil }
   Write ('n wird initialisiert: ');
   n:= 99;
   Ausgabe
END.
```

Ein Hauptprogramm dazu könnte so aussehen:

```
PROGRAM Haupt2;

USES Beispiel2;

BEGIN
   n:= n+1;
   Write('Neuer Wert von n: ');
   Ausgabe
END.
```

Das Unit hat also am Ende einen globlalen Anweisungsteil, der mit einem „END" endet (sonst enden Units ja mit einen „END" ohne vorangehenden Anweisungsteil). Diese Anweisungsfolge wird der Initialisierungsteil des Units genannt und automatisch vor dem Start des Hauptprogramms, das dieses Unit benutzt, ausgeführt.

Also erzeugt unser kleines Programm folgende Ausgabe:

```
n wird initialisiert: n = 99
Neuer Wert von n: n = 100
```

Intern wird das so gehandhabt, daß der Init-Teil als Prozedur mit dem qualifizierten Namen „_init" an den Linker exportiert wird, in unserem Beispiel also „Beispiel2._init". Deshalb können Sie im Interfaceteil eines Units keine Prozedur, Funktion oder Variable mit dem Namen „_init" deklarieren, denn dann würde der Linker ja eine Bezeichnerkollision feststellen und als Fehler melden.

Der Aufruf von Initialisierungsteilen erfolgt im Zweifelsfall genau so, wie man sich das wünscht. Ich möchte nur deshalb genauer darauf eingehen, damit Sie nicht sagen können, das sei nirgendwo dokumentiert:

1. Der Initialisierungsteil jedes direkt oder auch indirekt benutzten Units eines Programms wird genau einmal aufgerufen, auch dann, wenn ein Unit mehrfach benutzt wird (indem es von mehreren benutzten Units ge-used wird).
2. Benutzt ein Unit A ein anderes Unit B, so sorgt MaxonPASCAL dafür, daß in jedem Fall der Initialisierungsteil von Unit B vor dem von Unit A aufgerufen wird. Sie können also beim Programmieren eines Init-Teils davon ausgehen, daß alle Units, die Sie in Ihrem Unit benutzen, bereits initialisiert sind.
3. Von 2. einmal abgesehen, werden Units in der Reihenfolge winitialisiert, in der sie in der USES-Anweisung stehen.

9.11.2 Prüfsummen

Wie Sie bereits wissen, wird der Compiler durch eine USES-Anweisung veranlaßt, die Interface-Datei des entsprechenden Units zu lesen. Dort erfährt er, welche Bezeichner in das gerade übersetzte Unit oder Programm importiert werden, und vor allem auch, welche Bedeutung diese Bezeicher haben. So weiß er nach „USES Beispiel2" (siehe oben), daß „n" eine Integer-Variable und „Ausgabe" eine Prozedur ohne Parameterliste ist. Mit diesem Wissen wird nun der Rest des Programms übersetzt. So weit, so hoopy. Aber jetzt kommt's:

```
UNIT A;

VAR n: LongInt;

IMPLEMENTATION

BEGIN
```

```
    n:= 26731
END.
```

Dieses kleine Unit werde nun von einem zweiten benutzt:

```
UNIT B;

USES A;

PROCEDURE Ausgabe;

IMPLEMENTATION

PROCEDURE Ausgabe;
   BEGIN
     Writeln(n)
   END;

END.
```

Dies alles ist noch völlig klar und einsichtig: Das Unit exportiert eine Prozedur, die eine vom Unit „A" importierte LongInt-Variable ausgibt. Aber jetzt (die Spannung steigt ins Unerträgliche...) kommt das Hauptprogramm dazu:

```
PROGRAM C;

USES B;

BEGIN
   Write('Wert von n: ');
   Ausgabe
END.
```

Auch jetzt haben wir noch nicht den springenden Punkt erreicht. Der kommt erst, wenn wir das Unit „A" ändern: Wir deklarieren die Variable „n" im Interface-Teil nicht mehr als LongInt, sondern nur als einfaches Integer, und compilieren dieses Unit mit dieser Änderung.

Und nun (und damit wären wir endlich beim Thema) versuchen wir erneut, unser Hauptprogramm zu übersetzen.

„Na und?"

Das hüpfende Komma (oder so ähnlich) ist, daß das Unit „B" nach der Änderung von „A" noch nicht wieder neu übersetzt wurde. Ergebnis: der vom Compiler erzeugte Maschinencode jenes Units glaubt immer noch, „n" sei eine LongInt-Variable. Resultat: Die Prozedur „Ausgabe" liest ein Langwort (LongInt = 4 Bytes) aus dem Speicher und gibt es aus, obwohl dort nur ein Wort (Integer = 2 Bytes) für die Variable reserviert und initialisiert sind. Ergebnis: Es wird ein undefinierter (also im Zweifelsfall unsinniger) Wert ausgegeben, und Sie werden bei der Suche nach dem vermeintlichen Bug halb wahnsinnig (falls wir einfach 'mal so tun, als ob die Units und das Hauptprogramm jeweils so ungefähr tausend Zeilen lang und entsprechend unübersichtlich wären).

Aber keine Panik: das Ganze passiert nämlich absolut NICHT! MaxonPASCAL ist clever genug, das Problem zu erkennen, und meldet beim Compilieren des Programms „C" in der USES-Zeile:

```
Checksum Error: Unit "B" using different "A".
```

Alles klar: Das Unit „B" benutzte, als es compiliert wurde, ein Unit „A", das sich von dem jetzt vorliegenden unterscheidet. Nun wissen wir also, daß das Unit „B" neu compiliert werden muß. Am bequemsten benutzt man dazu das Pull-Down-Menü „Starten/Compile Datei" bzw. das Tastaturmenü „eXternal".

Obige Fehlermeldung enthält auch einen Hinweis darauf, woran der Compiler den Fehler benerkt hat: Er legt zu jedem Unit eine Prüfsumme an. Dann werden im Interfacefile jedes Units die Prüfsummen aller benutzten anderen Units vermerkt. Beim Lesen des Interfaceteils von Unit „B" stellte MaxonPASCAL fest, daß das hier benutzte Unit „A" nach dem letzten Übersetzen von Unit „B" verändert wurde.

Die Prüfsumme (32 Bit breit) wird übrigens nur über den Interface-Teil eines Units gebildet. Es hätte also keine Folgen gehabt, wenn Sie nur im Implementationsteil von Unit „A" etwas geändert hätten - denn das ist ja sowieso Privatsache dieses Units. Ferner gehen nur die wesentlichen Teile des Interfaceteils in die Prüfsumme ein, also keine Kommentare, Leerzeilen und andere Layout-Details.

9.12 Projekte - Modulares Programmieren mit Units

9.12.1 Aufräumen im Unit-Verzeichnis

Units sind, wie Ihnen nicht entgangen sein dürfte, ein sehr leistungsfähiges Konzept, das Sie sicher oft benutzen werden. Auf die Dauer wird sich da einiges in Ihrem Unit-Verzeichnis ansammeln: Standard-Units, selbstgeschriebene Units, diverse Units für alle möglichen Zwecke, die Sie auf Ihren MaxonPASCAL-Disketten oder in anderen Quelltextsammlungen gefunden haben usw. - und zwar jeweils zwei Dateien pro Unit. Somit dürfte Ihr Unit-Verzeichnis im Laufe der Zeit ganz schön voll werden. Irgendwann riskieren Sie dann auch, die Übersicht zu verlieren und einem Unit einen bereits vorhandenen Namen zu geben, wodurch das andere Unit überschrieben würde.

Was können Sie also gegen das drohende Chaos unternehmen (einmal davon abgesehen, daß Sie nicht mehr benötigte Units löschen können und sollen)? Seit dem Dahinscheiden von CP/M (die älteren unter Ihnen werden sich vielleicht noch daran erinnern - auch ich habe z. B. auf einem unter CP/M laufenden AppleII-Clone PASCAL gelernt, damals, in der guten alten Zeit... aber lassen wir das!) bietet jedes real existierende Betriebssystem (die 1541 und andere immer noch produzierte Fossilien übergehen wir hier einmal - so etwas KENNT man als AMIGA-User gar nicht) eine Möglichkeit, Verzeichnisse aller Art aufzuräumen: Richtig, Unterverzeichnisse sind gemeint („Sub-Directories" in Basic German). Nichts liegt also näher, als auch im Unit-Verzeichnis damit klar Schiff zu machen...

Das Problem: woher soll MaxonPASCAL wissen, wohin es Units schreiben bzw. von wo lesen soll? (Naive Menschen glauben immer noch, mit einem Computer Probleme lösen zu können. In

Wirklichkeit zieht die Anschaffung eines Computers generell unzählige Probleme mit sich, von denen man vorher nicht einmal zu träumen gewagt hätte...) Aber Sie haben Glück: in diesem Abschnitt erfahren Sie nicht nur das Problem, sondern auch die Lösung (Wow!), und zwar genau jetzt (Oh yeah!):

Zunächst müssen wir unser Unit ja in ein Unterverzeichnis schreiben.Dazu dient die Compiler-Directive „$project", die man an beliebiger Stelle im Unit-Quelltext unterbringen kann, z. B. so:

```
{$project MatheTools }
```

Der Projektname muß formal ein Bezeichner sein und ist die Bezeichnung des Unterverzichnisses, in dem das Unit abgelegt werden soll. Dabei wird das entsprechende Verzeichnis automatisch angelegt.

Man kann aber natürlich auch „manuell" in Ihrem Unit-Directory solche Subdirectorys anlegen und Units dort nachträglich ablegen. In Programmen und Modulen darf man „$project" zwar auch verwenden, aber es hat dort keinerlei Wirkung. Als Abkürzung kann man auch „$pro" oder alles, was mit „$proj" anfängt, schreiben.

Um ein Unit aus einem Unterverzeichnis des Unit-Verzeichnisses zu benutzen, setze man einfach ein „FROM <Name>" vor die Uses-Zeile, z. B. so:

```
USES Intuition, Graphics;
FROM MatheTools USES LongMath, Matrix;
FROM Windows USES PASCALCrt;
USES Printer;
```

Mit diesen Zeilen werden die Units „Intuition", „Graphics" und „Printer" aus dem Unit-Hauptdirectory sowie „LongMath", „Matrix" und „PASCALCrt" aus den Unterverzeichnissen „MatheTools" bzw. „Windows" gelesen. Das FROM bezieht sich also immer nur auf das nächste USES.

Für Modula-II-Kenner: Die MaxonPASCAL-Zeile

```
FROM xxx USES yyy;
```

hat nichts mit der Modula-Anweisung

```
FROM xxx IMPORT yyy;
```

zu tun! Ersteres importiert ein ganzes Unit aus einem Subdirectory, letzteres einen einzigen Bezeichner aus einem Modul.

9.12.2 Organisation von Projekten

Units dienen zwei verschiedenen Zwecken: Wie Sie sich Bibliotheken mit öfter benutzten Prozeduren und Funktionen anlegen können, wissen Sie bereits. Aber Sie können mit Units auch modular programmieren, und zwar komfortabler als mit MODULEn - wenn auch mit ein paar klei-

nen Einschränkungen. Formal besteht natürlich keinerlei Unterschied zwischen „Bibliotheks"- und „Modul"-Units, erstere werden eben von vielen verschiedenen Programmen benutzt, während letztere speziell für ein Programmprojekt geschrieben werden. Trotzdem will ich hier einige Tips für die Realisierung umfangreicher Programmprojekte unter Benutzung von Units geben.

Als erstes sollte man sich einen Namen für das Projekt ausdenken, denn Units, die nur in diesem einem Programm benutzt werden, haben im Unit-Hauptverzeichnis nichts zu suchen. Auch für die Quelltexte ist entweder eine neue Diskette zu formatieren oder ein eigenes Unterverzeichnis anzulegen.

Der nächste Job ist schon etwas anspruchsvoller: Man muß ein gewisses Konzept entwickeln, nach dem man sein Projekt in Units unterteilen will. In der Regel fallen einem bei großen Programmen sofort gewisse Funktionsgruppen auf, die weitgehend unabhängig voneinander arbeiten.

Ein Dateiprogramm könnte man zunächst in folgende Units aufteilen:

- Festlegen der Eingabemaske
- Eingabe eines Datensatzes
- Lesen und Schreiben einer Datei
- Prozeduren zum Drucken von Listen usw.

Jedes dieser Units könnte man noch weiter verfeinern. Das sollte man aber nicht unbedingt übertreiben, z. B. nicht für jede Prozedur ein eigenes Unit. Ein einzelnes Unit sollte zweckmäßigerweise nicht mehr als etwa 1000 Zeilen Quelltext umfassen.

Beim Entwurf dieses Konzeptes ist zu beachten, daß Units sich nicht gegenseitig benutzen können, hier also eine strenge Hierarchie besteht. Deshalb wird man des öfteren mehreren der Hauptunits ein gemeinsames Unterunit zuordnen müssen. Oft ist es sogar zweckmäßig, ein Unit namens „Global" (oder so) zu schreiben, das globale Definitionen (Variablen, Datentypen...) des Programms enthält und von allen anderen Units benutzt wird.

Dann brauchen Sie noch ein Hauptprogramm. Es sollte aus gewissen Gründen (nicht Gewissensgründen) so kurz wie eben möglich sein, im Extremfall beispielsweise bloß ein Unit „Haupt" über USES einbinden und daraus eine Prozedur namens „Hauptprogramm" aufrufen, der ganze Rest könnte dann in jenem Unit passieren.

Ein Programm (in diesem Kapitel heißt das immer: „ein riesiges Programmprojekt") schreibt man natürlich nicht einfach so 'runter. Vielmehr muß es immer wieder getestet, erweitert, getestet, erweitert, getest... werden. Wenn Sie aber versuchen, ein Unit zu starten, wird es übersetzt, der Code ins Unit-Verzeichnis geschrieben und die monotone Meldung „Can't run a unit!" ausgegeben - auf gut deutsch: Ein Probelauf eines Units ist nicht möglich. Deshalb braucht man das Hauptprogramm, und deshalb sollte es auch möglichst kurz sein: Immer, wenn man ein Unit geändert und compiliert hat, muß man das Hauptprogramm durch den Compiler jagen, so daß man das gesamte Programm mit dem geänderten Unit ausprobieren kann.

Es wäre ein unzumutbarer Arbeitsaufwand, wenn man dazu jedesmal erst den Quelltext des gerade bearbeiteten Units abspeichern und das Hauptprogramm in den Editor laden müßte. Einfacher geht

es, wenn man das Hauptprogramm extern übersetzen läßt, also mit dem Menü „Starten/Compile Datei" oder dem entsprechenden Tastaturmenü. Auf diesem Wege kann man übrigens auch Units übersetzen, die wegen eines Prüfsummenfehlers neu compiliert werden müssen.

Noch komfortabler wird es aber, wenn man die Quelltextdatei des Hauptprogramms als „Main File" bzw. „Hauptdatei" definiert. So eine Datei wird immer, nachdem aus dem Editor ein Unit compiliert wurde, extern übersetzt und gelinkt. Also wird nach jeder Änderung eines Units das Gesamtprogramm neu erstellt und kann getestet werden.

9.13 Assembler und PASCAL

Es gibt Tage im Leben eines Programmierers, da kann man es einfach nicht vermeiden, ein Unterprogramm in Assembler zu schreiben; sei es nun aus Geschwindigkeitsgründen oder bei maschinennahen Problemen (versuchen Sie 'mal, einen Exceptionhandler oder Interruptserver in PASCAL zu schreiben...). Der MaxonPASCAL-Linker bindet nicht nur PASCAL-Module zusammen, sondern kann auch Assemblerroutinen einlinken.

Prinzipiell können Sie jeden Assembler benutzen, der Alink-kompatible Objektdateien erzeugt (also fast jeden). Ich empfehle natürlich den MaxonASM.

Assemblerroutinen werden genau wie PASCAL-Module eingebunden. Sie können auch umgekehrt vom MaxonASM aus das PASCAL-Hauptprogramm einlinken, indem Sie das entsprechende Objectfile und die Laufzeitbibliothek „paslib.o" hinzulinken.

Ein Unterprogramm wird aus dem MaxonASM exportiert, indem man das Label am Anfang der Routine als „xdef" deklariert. Auch Variablen kann man auf diese Weise zwischen der Assembler-Routine und dem PASCAL-Programm austauschen.

Ein Programmfragment als Beispiel:

```
Program PASCAL_meets_Assembler;
Var a, b: integer; EXPORT;
      x: LongInt; IMPORT;
Procedure Assi;    IMPORT;
Begin { Hauptprogramm }
   a:= 42;
   b:= 17;
   Assi;
   writeln(a:10,b:10,x:10)
End.
```

Ein geeignetes MaxonASM-Programm dazu:

```
        xdef a,b,x,Assi
        xref _main
        jmp _main                ; Einsprung ins Hautprogramm

Assi:   move.w  a,d0
        move.w  b,a
```

```
        move.w  d0,b    ; Werte von "a" und "b" vertauschen
        muls    a,d0        ; Produkt a*b berechnen
        move.l  d0,x        ; ...und in x ablegen
        rts                     ; zurück ins PASCAL-Programm

x:      blk.l   1               ; Langwort-Variable, wird erportiert
```

Dieses Programm macht nicht viel Tiefsinniges: Zwei INTEGER-Variable werden initialisiert, eine Assembler-Routine, die die Variablenwerte vertauscht und das Produkt in einer LONG-Variablen ablegt, wird aufgerufen, und am Ende werden alle drei Variablenwerte ausgegeben.

Wie kann man nun eine Parameterübergabe realisieren? Es gibt natürlich - wie in obigem Beispiel - die Möglichkeit, die Parameter in gemeinsamen Variablen abzulegen. Wenn Sie einer importierten Routine wie einer PASCAL-Prozedur Parameter übergeben, liegen diese direkt oberhalb der Rückkehradresse auf dem Stack (also ab „4(a7)"), wobei die einzelnen Parameter in ihrer normalen Reihenfolge auf den Stack geschoben wurden, so daß also der erste Parameter an einer höheren Speicheradresse steht als der zweite.

Der Rückgabewert von Functions ist stets im Register d0 zurückzugeben. Ist er vom 64 Bit breiten Typ „LongReal", so ist er in d0 und d1 abzulegen. Der Wert des Adressregisters a5 darf in der Assembler-Routine nicht verändert werden, davon abgesehen stehen Ihnen alle Register zur Verfügung.

Einfacher geht die Parameterübergabe aber, wenn Sie im PASCAL-Programm „External" statt „Import" schreiben. Die Wirkung ist eigentlich dieselbe; wenn aber ein Parameter einen Registernamen wie „d0" oder „a3" trägt, wird der Wert vor dem Unterprogrammaufruf in das entsprechende Register geladen. Bei VAR-Parametern erhält das Register die Adresse der referierten Variablen, bei nichtskalaren Value- Parametern (z. B. Arrays) ist das Ergebnis undefiniert.

Die Parameterübergabe in Registern hat auf die Laufzeit übrigens keine Auswirkung, denn die Parameter werden zuerst wie gewohnt auf den Stack gelegt und erst dann in die Register geladen. Wenn ein formaler Parameter keinen 68000-Registernamen trägt, wird er auch nicht in ein Register geladen, liegt aber wie gewohnt auf dem Stack.

Die Assembler-Unterroutine kann wie ein Modul mit der Anweisung „{$link 'xxx'}" in das PASCAL-Programm eingebunden werden. Dann haben Sie aber möglicherweise Probleme mit dem Pfad, der sich beim Aufräumen auf der Diskette ändern kann. Deshalb gibt es alternativ zu „link" noch die „ulink"-Anweisung.

Beispiel:

```
{$ulink "Assemblerteil.o" }
```

Jetzt wird die angegebene Datei aus dem Unit-Verzeichnis gelesen. Auch hier können Sie Unterverzeichnisse angeben (analog zum „FROM" bei Units):

```
{$ulink "Assemblerzeugs/Test.o" }
```

„ulink" hat noch einen weiteren Vorteil gegenüber „link": Bei Angabe zweier Unit-Pfade kann man mit „ulink" wie bei Units ein Puffern der Objektdateien in der RAM-Disk erreichen. Dann müssen Sie aber aufpassen, wenn Sie gleichzeitig in PASCAL und Assembler programmieren und eine Assembler-Objektdatei ändern: vergessen Sie nicht, die neue Version der Objektdatei in BEIDEN Unit-Verzeichnissen abzulegen, damit MaxonPASCAL sie auch sicher findet.

Stehen „$link" und „$ulink" im Interfaceteil eines Units, so werden sie auch in die Interfacedatei übernommen, so daß die Dateien dann beim „USES"-Aufruf des Units automatisch eingeladen werden.

10. MAKE - die integrierte Projektverwaltung

10.1 Das Prinzip

10.1.1 Wozu eine Projektverwaltung?

Größere Projekte bestehen normalerweise nicht aus einer einzelnen Quelltextdatei, sondern aus einer Vielzahl von Modulen, die vom Linker verbunden werden - zumindestens sollte es so sein. Das nimmt leicht Ausmaße an, die nicht mehr ganz leicht zu überschauen sind. Hinzu kommt, daß es natürlich Unsinn ist, nach jeder Änderung das ganze Projekt zu übersetzen. Auf die Dauer wird es aber lästig, sich immer wieder neu zu überlegen, welches Modul jeweils übersetzt werden muß, einmal ganz davon abgesehen, daß man eigenartige Überraschungen erleben kann, wenn man dabei etwas vergißt. So entsteht das Bedürfnis, solche Vorgänge zu automatisieren - wozu hat man schließlich einen Computer...

Die populärste Projektverwaltung, die auf so ziemlich allen ernstzunehmenden Rechnern anzutreffen und bei allen Compilersprachen tausendfach bewährt ist, heißt „make". Eigentlich ist „make" selbst eine Art Programmiersprache, und das macht es natürlich nicht leicht, ein komplettes Make in eine graphisch orientierte Entwicklungsumgebung wie die von MaxonPASCAL zu integrieren. Deshalb unterstützt MPASCAL nur einen recht begrenzten (aber den entscheidenden) Teil von „make". Außerdem beschränkt „Make" sich auf Projekte, die aus einem Hauptprogramm und einigen Modulen bestehen. Durch Units hierarchisch gegliederte Projekte werden also nicht unterstützt, auch wenn man natürlich in seinem Projekt auch Units benutzen und dazulinken kann. Units werden aber - im Gegensatz zu Modulen - nicht von der Projektverwaltung automatisch compiliert.

10.1.2 Was ist das überhaupt - ein Projekt?

Fangen wir doch einmal ganz graswurzelmäßig von vorn an: Woraus besteht ein Projekt (immerhin war von einer Projekt-Verwaltung die Rede) - und was ist überhaupt ein Projekt?

Also, zu den oben erwähnten Einschränkungen gehört, daß ein Projekt immer genau ein ausführbares Programm darstellt. Wenn man im Rahmen eines Projekts ein Programm in mehreren Versionen erstellen will, muß man eben für jedes einzelne Executable eine eigene Projektdatei erstellen.

OK - ein Projekt ist also ein ausführbares Programm. Und wie kommt man zu einem solchen Programm, sofern es sich nicht gerade um ein Bonsai-Miniatur-Projekt handelt?

Richtg, das erwähnte ich schon: Man linkt dafür Objektdateien, also Module, zusammen. Genaugenommen bindet der Linker auch noch fertige Bibliotheksmodule ein, aber das interessiert uns hier nur höchst periphär, oder genaugenommen nicht die Bohne. Der Freizeit-Existenzialist stellt sich jedenfalls nun die Frage „Und wo kommen die kleinen Module her?", der Coder weiß es schon: Das hat ausnahmsweise nichts mit Schweinskrams zu tun, sondern die Objektdateien werden von irgendwelchen Compilern erzeugt.

Zu jedem der Module gibt es genau einen ganz besonderen Quelltext, nämlich den Text, aus dem die Objektdatei durch Compilieren entsteht. Um so ein Objekt zu machen, braucht die Projektverwaltung also im wesentlichen eine Information darüber, wie der Quelltext dazu heißt, und außerdem noch ein paar Zusatzinformationen, etwa den Namen des Compilers oder irgendwelche der allseits beliebten Compiler-Optionen.

Aber das ist noch lange nicht alles, denn zu einem normalen Projekt gehören noch einige Texte mehr - normalerweise Includedateien, die von den eigentlichen Quelltexten irgendwo und irgendwie benutzt werden. Das führt uns natürlich zu der Frage, wann und wie eine Objektdatei neu übersetzt wird. Nun, bevor wir alle in Rente gehen, werden wir vielleicht noch das Entwicklungssystem erleben, das Programme on-line compiliert, während man den Quelltext bearbeitet. Vorerst ist es jedenfalls noch so, daß man munter an seinen Quelltexten 'rumediert und irgendwann entscheidet, daß das Programm für eine erneute Übersetzung reif ist. Dann tritt die bereits mehrfach erwähnte Projektverwaltung „Make" in Aktion und schaut nach, welche Objektdateien für das Projekt nötig sind, und bevor diese zusammengelinkt werden, denkt es scharf darüber nach, welche der Objektdateien neu übersetzt („recompiliert") werden muß. Da ja der Compiler wenigstens zeitweise mit dem Programmierer Schritt halten soll, ist auch noch halbwegs einsichtig, wann das zu geschehen hat: Richtig, genau dann, wenn mindestens einer der Texte, aus denen das Modul entsteht (d. h. Quelltext und Include-Files), seit der letzten Übersetzung verändert wurde. Das läßt sich leicht an Erstellungsdatum und -uhrzeit der entsprechenden Dateien feststellen.

10.1.3 Was hat das ganze mit „Make" zu tun?

Die Projektverwaltung verwaltet im wesentlichen zweierlei Datenbestände: Texte und Objekte. Bei den Texten handelt es sich im allgemeinen um irgendwelche Quelltexte oder Includedateien, und aus diesen Texten erzeugt der Compiler unter der Kontrolle der Projektverwaltung die Objektdateien. Im letzten Schritt werden diese Objektdateien dann mit den Libraries zu einer ausführbaren Datei gelinkt.

Für jede der Objektdateien braucht das Make also Angaben darüber, wie es „gemacht" (daher der Name) werden soll, insbesondere den Namen eines Quelltexts und eines Compilers. Das würde man natürlich auch noch von Hand hinkriegen, und deshalb sagt man der Projektverwaltung auch, unter welchen Umständen ein Objekt compiliert werden soll und wann die alte Version noch einmal verwendet werden kann.

Wie bereits gesagt, muß ein Objekt neu übersetzt werden, wenn ein Text, der mit diesem Objekt zu tun hat, verändert wurde. Normalerweise handelt es sich bei diesen Texten um den Quelltext des Moduls sowie die darin benutzten Includedateien. Dazu vergleicht Make den Datestamp, also die Zeit der letzten Veränderung einer Datei, der Objektdatei mit denen „ihrer" Texte. Natürlich wird ein Objekt auch dann compiliert, wenn es bisher noch nicht existierte.

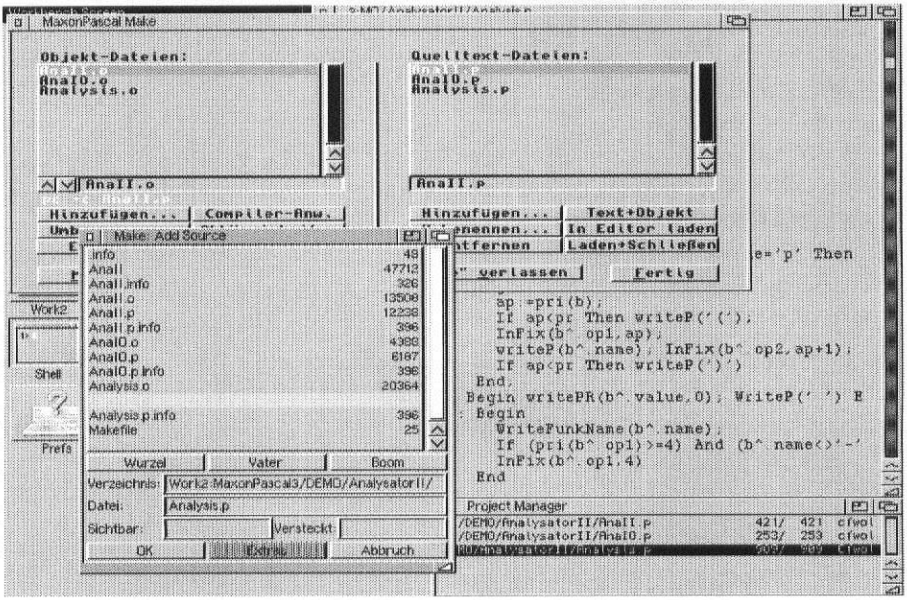
10.2 Die Praxis

10.2.1 Erstellen von Projekten

Alle Angaben über die Struktur eines Projekts werden sinnvollerweise in einer Datei festgehalten, dem Makefile. Mit MaxonPASCAL erstellen Sie diese Datei interaktiv mit dem Pull-Down-Menü **„Make"** und dem Make-Dialogfenster. Das Makefile wird dabei ständig aktualisiert, also nach jeder Veränderung neu abgespeichert. Da eine solche Datei normalerweise nicht sehr lang sein wird, dürfte dabei auch beim Diskettenbetrieb keine Verzögerung auftreten.

Am besten probieren Sie das Make-Utility einfach aus, indem Sie sich ein Mini-Projekt aus zwei Modülchen erstellen. Damit fangen Sie an, indem Sie den Menüpunkt **„Neues Makefile"** anwählen. Wenn bisher keine Projektdatei geladen ist, können Sie ebenso gut auch den Menüpunkt **„Projektverwaltung"** benutzen. Nun erscheint der Filerequester und fragt Sie nach dem gewünschten Dateinamen. Meist nennt man die Datei ganz einfach „makefile", jedenfalls dann, wenn man nicht mehr als eine solche Datei in einem Verzeichnis hat. Wenn Sie dagegen an einem vorhandendem Projekt weiterarbeiten wollen, benutzen Sie den Menüpunkt **„Makefile laden"**.

Nach dem Erzeugen einer neuen Projektdatei öffnet sich ein Fenster, das zwei Scrollisten (eine mit „Objekt-Dateien", die andere mit „Quelltext-Dateien" betitelt) und eine ganze Menge Gadgets enthält. Klicken Sie hier am besten einfach auf das Gadget ***„Fertig"*** in der rechten unteren Ecke, um das Fenster wieder zu schließen.



An einem Projekt, daß mehr als einen Quelltext umfaßt, wird man in der Regel länger als einen Tag lang arbeiten. Deshalb sollten Sie die folgenden Tips beachten, um es dabei auf lange Sicht komfortabel zu haben:

☞ Zu jedem Projekt gehört ein eigenes Verzeichnis. Sie sollten also auf jeden Fall mit der Workbench oder Shell ein neues Unterverzeichnis in Ihrem „PASCAL:"-Verzeichnis oder auch anderswo anlegen.

☞ Wenn Sie MPASCAL immer von der Shell starten, sollten Sie vorher stets in das Verzeichnis wechseln, in dem das Makefile liegt. Dadurch wird dies auch für MPASCAL und Edward zum aktuellen Verzeichnis, und dadurch werden die SESSION-Dateien von dort gelesen bzw. dahin geschrieben. Insbesondere werden dann auch die Compiler-Einstellungen von MPASCAL im passenden Verzeichnis abgelegt und beim nächsten Start von MPASCAL automatisch wieder eingeladen.

☞ MPASCAL kann von der Workbench durch Anklicken des Makefile-Icons gestartet werden. Die Projektdatei wird dabei automatisch als solche erkannt und geladen (und nicht etwa als Textdatei in den Editor). Dabei wird das Verzeichnis, in dem das Makefile liegt, zum aktuellen Verzeichnis von MPASCAL und Edward, was wiederum erfreuliche Auswirkungen auf die Behandlung der SESSION-Files hat (siehe oben).

Die bequemste Art, sich ein Projekt für MaxonPASCAL einzurichten, geht deshalb so:

1. Stellen Sie sicher, daß die Option „Icons erzeugen" eingeschaltet ist und daß ein geeignetes leeres Verzeichnis für Ihr Projekt existiert.
2. Erzeugen Sie in diesem Verzeichnis wie oben beschrieben eine Make-Datei, die Sie zunächst einmal so leer lassen, wie sie ist.
3. Beenden Sie jetzt MPASCAL.
4. Wenn Sie Workbench 2.0 (oder später) benutzen, machen Sie jetzt am besten ein „Leave Out" (auslagern) des Makefile-Icons. Dadurch liegt es immer sichtbar auf dem Workbench-Bildschirm und kann nach dem Einschalten des Rechners sofort geöffnet werden.
5. Starten Sie nun MPASCAL, indem Sie das Makefile-Icon (natürlich doppelt) anklicken. Nun werden „Edward.SESSION" und „mp.SESSION" automatisch aus dem Verzeichnis des Projekts gelesen (bzw. dorthin geschrieben).

Vielleicht erscheint Ihnen diese Prozedur etwas umständlich, aber auf die Dauer werden Sie es zu schätzen wissen, nach dem Booten des Rechners mit einem einfachen Doppelklick genau da weiterarbeiten zu können, wo Sie zuletzt aufgehört haben.

10.2.2 Hinzufügen von Dateien, umständliche Methode

Zu einem Projekt gehört natürlich mindestens eine Objektdatei, und um eine solche zu erzeugen, braucht man einen Quelltext. Versuchen wir es zunächst auf die umständliche Methode und schreiben einen Text, z. B.

```
module testmodul;
procedure test; export;
        begin
                writeln('Dies ist ein Test.')
        end;
begin
end;
```

Diesen Text speichern Sie einmal als „test.p" ab (mit einer Datei ohne Namen kann Make nichts anfangen) und wählen dann den Menüpunkt **„Make/Projektverwaltung"**. Damit öffnen Sie wieder das Make-Fenster, das Sie schon beim Erzeugen der Projektdatei gesehen haben. Dabei ist Ihnen vielleicht auch schon aufgefallen, daß das Fenster vertikal zweigeteilt ist, wobei die linke Hälfte mit „Objekt-Dateien" und die Rechte mit „Quelltext-Dateien" überschrieben ist.Wenn Sie nun die Textdatei „test.p" und die Objektdatei „test.o" in Ihr Projekt aufnehmen möchten, geschieht dies eigentlich ganz intuitiv: Klicken Sie zunächst auf das Gadget ***„Hinzufügen..."*** in der rechten Spalte. Es öffnet sich nun ein Dateiauswahlfenster, in dem Sie die soeben erzeugte Datei „test.p" anwählen. Sie bestätigen die Auswahl, der Filerequester wird geschlossen und schon steht „test.p" in der rechten Liste des Make-Fensters. Ebenso einfach wählen Sie jetzt das linke Gadget ***„Hinzufügen..."*** an und wählen im Filerequester den Dateinamen „test.o". Höchstwahrscheinlich existiert diese Datei noch überhaupt nicht, so daß Sie sie folglich nicht einfach anklicken können, sondern den Namen von Hand eintippen müssen.

Natürlich spielt es keine Rolle, ob sie zuerst - wie hier beschrieben - die Quelltextdatei und dann die Objektdatei oder umgekehrt eintragen. Die beiden Listen sind zunächst vollkommen voneinander unabhängig, und wenn Sie möchten, können Sie jetzt auf beiden Seiten beliebig viele Dateien eintragen, die allesamt auf der Platte vorhanden sein können, aber nicht müssen. Die Dateien müssen auch keineswegs im selben Verzeichnis wie die Make-Datei liegen, aber das ist allgemein üblich und wohl auch am praktischsten.

Wenn Sie jeweils mehrere Dateien aufnehmen, wird Ihnen auffallen, daß die Namen der Quelltext-Dateien alphabetisch sortiert werden, während die Objektdateien in der Reihenfolge erscheinen, in der Sie hinzugefügt werden. Dies ist auch die Reihenfolge, in der der Linker später die Objektdateien aneinanderhängt, was z. B. dann von Interesse ist, wenn man einen eigenen Startup-Code schreiben möchte. Die Reihenfolge der Objektdateien können Sie mit den beiden Pfeilgadgets links von der Anzeige für die angewählte Objektdatei verändern.

Dumm wie Computer nun einmal sind, müssen Sie nun aber auch noch klarmachen, daß die Objektdatei „test.o" entsteht, indem „test.p" compiliert wird. Wählen Sie dazu „test.o" in der linken Spalte an (so daß der Name invertiert dargestellt wird und in der Anzeige unter der Objekt-Liste

erscheint) und klicken Sie dann auf das Gadget „Compiler-Anw.". Alternativ können Sie „test.o" auch mit einem Doppelklick anwählen.

Nun öffnet sich ein neues Fenster, in dem alle Gadgets bis auf ein Cycle-Gadget oben links und die üblichen ***„OK"***- und ***„Abbruch"***-Gadgets am unteren Rand deaktivert sind. Das Cycle-Gagdet zeigt zunächst ***„Nicht übersetzbar"*** an. Nach einem Klick erscheint hier ***„intern compilieren"***, und alle Gadgets bis auf ein Stringgadget unten links werden gleichzeitig aktiv. Insbesondere kann jetzt auch die mit „Quelldatei" überschriebene Liste benutzt werden. Dort wählen Sie jetzt einfach „test.p" als Quelltext an und schließen das Fenster wieder mit einem Klick auf ***„OK"***.

Im Make-Fenster erscheint jetzt direkt unterhalb der Objekt-Liste „pc -c test.p". Dies ist eine Pseudo-Anweisung, mit der man in der Shell „test.p" nach „test.o" übersetzen würde, wenn MaxonPASCAL keine integrierte Entwicklungsumgebung hätte.

10.2.3 Übersetzen des Projekts

Sie können das Programm jetzt compilieren, indem Sie entweder im Make-Fenster auf das Gadget ***„Make"*** klicken oder wie gewohnt im Editor die Taste <F9> bzw. das Menü **„Compiler/Übersetzen"** anwählen. Nun tritt Make in Aktion, stellt fest, daß „test.o" compiliert werden muß, und bindet anschließend aus diesem Objectfile ein Programm zusammen - oder versucht es zumindest, denn es gibt ja noch kein Hauptprogramm. Make funktioniert völlig unabhängig davon, welcher Text sich gerade im Editor befindet.

10.2.4 Hinzufügen von Dateien, einfache Methode

Beim folgenden Miniatur-Modul, welches das noch fehlende Hauptprogramm enthält, machen wir es uns etwas einfacher:

```
program haupt;

{$incl "test.h"}
      begin
            test;
      end.
```

Diesen Text speichern wir einfach als „main.p" ab und wählen den Menüpunkt **„Text und Objekt aufnehmen"**. Jetzt macht Make vollautomatisch folgendes:

- ☞ Die Datei „main.p" wird in die Liste der Texte aufgenommen.
- ☞ Das Objekt „main.o" wird dem Projekt hinzugefügt.
- ☞ Die Compileranweisung für „main.o" wird passend gesetzt.

Jetzt fehlt dem Projekt nur noch eins: Die Header-Datei „test.h", in der die Funktionen (na ja, hier ist es ja nur eine) des Moduls „test" deklariert.

Die Datei test.h enthält folgende Zeile:

```
procedure test; import;
```

also den Prototypen der Funktion. So, wenn Sie jetzt den Compiler starten, müßte das Projekt korrekt übersetzt und gelinkt werden.

Zuvor sollten Sie aber „test.h" noch als Quelltext dem Projekt hinzufügen (im folgenden Abschnitt verrate ich Ihnen auch, wozu das gut sein soll). Das können Sie wie gehabt vom Make-Fenster aus erledigen. Wenn Sie die Datei aber sowieso gerade im Editor haben, können Sie auch den Menüpunkt **„Make/Text aus Editor aufnehmen"** benutzen, und schon wird der aktuelle Text des Editors in die Quelltext-Liste aufgenommen.

10.2.5 Abhängigkeiten

Das so ziemlich wichtigste Feature der Projektverwaltung haben wir aber bisher nicht benutzt: Die Abhängigkeiten. Offensichtlich benutzt in unserem Beispiel „main.o" die Datei „test.h" als Includedatei und ist folglich davon abhängig. Immer, wenn „test.h" verändert wurde, sollte deshalb auch „main.o" sicherheitshalber neu übersetzt werden.

Öffnen Sie also wieder das Make-Fenster, wählen Sie in der linken Liste die Objektdatei „main.o" an und klicken Sie dann auf das Gadget ***„Abhängigkeiten"***. Dabei werden Sie bemerken, daß dieses Gadget invertiert bleibt, bis Sie es erneut anklicken. Nun können Sie in der Quelltext-Liste alle die Dateien anwählen, von denen „main.o" abhängig sein soll, also insbesondere „test.h". Korrekterweise können Sie auch „main.p" anwählen, aber bei interner Übersetzung weiß MaxonPASCAL von selbst, daß jede Objektdatei von ihrer Quelltextdatei anhängt. Jedenfalls werden Sie dabei merk, daß Sie jetzt in der rechten Spalte beliebig viele Zeilen gleichzeitig aktivieren und durch erneutes Anklicken wieder deaktiveren können. Dieser Modus bleibt erhalten, so lange das Gadget ***„Abhängigkeiten"*** invertiert ist. Sie können jetzt auch ganz einfach auf eine andere Objektdatei klicken, um deren Abhängigkeiten zu betrachten und zu manipulieren.

Im Grunde genommen ist dieses manuelle Setzen von Anhängigkeiten aber überhaupt nicht nötig. Es reicht völlig, wenn Sie im System-Dialogfenster (Menüpunkt **„Optionen/System"**) das vielversprechend mit ***„Make-Datei: Abhängigkeiten automatisch setzen"*** beschriftete Symbol anwählen. Wann immer nun eine Objektdatei des Projekts compiliert wird, merkt MPASCAL sich, welche Includedateien dabei gelesen wurden - natürlich nur die selbstgeschriebenen aus dem aktuellen Verzeichnis und nicht etwa die System-Includes, die über den im Compiler-Requester angegebenen Pfad eingebunden werden.

Anschließend werden die Abhängigkeiten der jeweiligen Objektdatei automatisch gesetzt, wobei nötigenfalls sogar neue Textdateien in das Projekt aufgenommen werden.

Deshalb beschränkt sich das Anlegen eines Projekts zusammengefaßt auf folgende Schritte:

- Erzeugen sie eine neue Projektdatei.
- Laden Sie alle „echten" Quelltextdateien, also keine Includefiles, einmal in den Editor und wählen Sie dann den Menüpunkt **„Text und Objekt aufnehmen"**.

☛ Starten Sie die Übersetzung, wobei Sie sicherstellen sollten, daß die Objektdateien allesamt noch nicht existieren. Dann werden selbsttätig alle Quelltexte durchcompiliert, die Includedateien in das Projekt aufgenommen und die Abhängigkeiten korrekt gesetzt.

10.2.6 Weitere Funktionen im Make-Fenster

10.2.6.1 Projektpflege

Es kommt hin und wieder vor, daß eine Quelltext- oder Objektdatei komplett überflüssig wird. Dann wählen Sie die betreffende Datei einfach auf der richtigen Seite des Make-Fensters an und klicken auf das Gadget ***„Entfernen"***. Nach einer Sicherheitsabfrage wird die Datei vollständig aus dem Projekt entfernt.

Ebenso kann es passieren, daß Ihnen ein Dateiname plötzlich nicht mehr ganz passend erscheint. Dies können Sie elegant erledigen, indem Sie den Dateinamen im Make-Fenster anklicken und dann das Gadget ***„Umbenennen..."*** benutzen (aber bitte das auf der richtigen Seite, denn auch diese Funktion gibt es gleichermaßen für Objekt- und Quelldateien). Es erscheint der übliche Filerequester, dem Sie nun den gewünschten neuen Namen anvertrauen. Beachten Sie aber bitte, daß diese Funktion die Datei ausschließlich in der Make-Datei umbenennt. Das eigentliche Umbenennen der Datei auf der Festplatte bzw. Diskette müssen Sie schon selbst erledigen.

10.2.6.2 Übersetzung erzwingen

Eigentlich entscheidet das Make ja selbständig und prinzipiell auch korrekt, welche Datei wann recompiliert werden soll. Manchmal möchte man dies aber auch erzwingen, z. B. wenn man Compileroptionen verändert hat und einige oder alle Objektdateien gezielt mit diesen neuen Optionen übersetzen lassen möchte. Deshalb gibt es im Make-Fenster das Gadget „Markieren".

Ähnlich wie ***„Abhängigkeiten"*** ist auch ***„Markieren"*** ein Umschaltbares Gadget. Während es aktiviert ist, können Sie in der Objektliste beliebig Dateien anwählen und deaktivieren. Beim nächsten Übersetzungsvorgang werden die so markierten Dateien dann auf jeden Fall compiliert, und zwar zusätzlich zu den Dateien, die Make ohnehin für übersetzungswürdig befindet.

10.2.6.3 Dateien in den Editor laden

Beim Programmieren hat man es fast immer nur mit den Textdateien zu tun, die zum aktuellen Projekt gehören. Deshalb bietet MPASCAL eine elegante Möglichkeit, Quelltexte in den Editor zu laden.

Öffnen Sie dazu das Make-Fenster und wählen Sie in der rechten Liste den gewünschten Quelltext an. Wenn Sie nun das Gadget ***„In Editor laden"*** anklicken, wird der Text - wer hätte das gedacht - in den Edward hinzugeladen. Das Gadget ***„Laden+Schließen"*** funktioniert genauso, schließt aber gleichzeitig das Make-Fenster. Dies können Sie auch mit einem Doppelklick auf den Dateinamen erreichen.

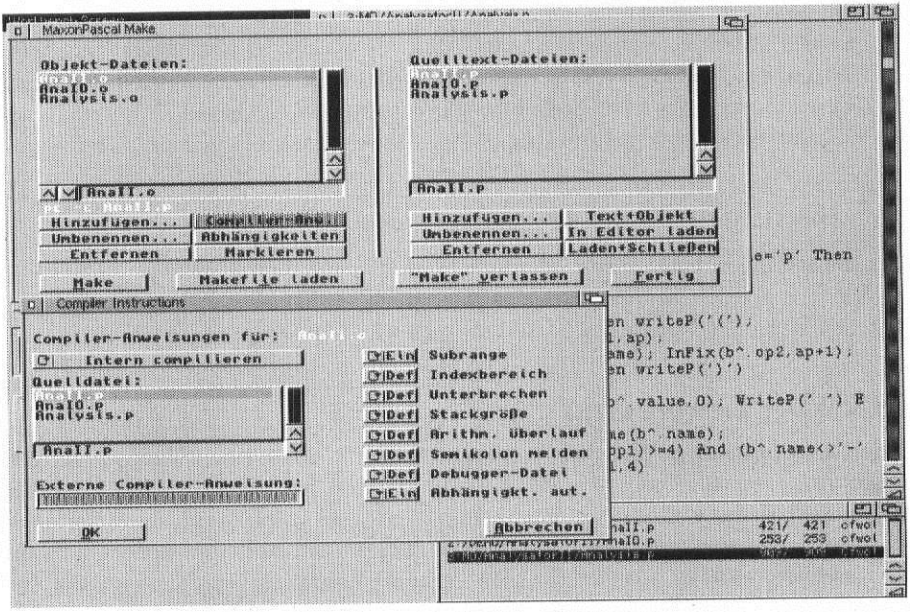
10.2.6.4 Projektdatei wechseln

Am unteren Rand des Make-Fensters befinden sich zwei bisher unerwähnte Gadgets: ***„Makefile laden"*** und ***„Make verlassen"***. Naheliegenderweise erscheint bei Anwahl des ersteren die Datei-Auswahlbox und ermöglicht Ihnen, eine andere Projektdatei zu laden. Mit dem anderen Gadget verlassen Sie den Make-Modus und gelangen wieder in die normale Betriebsart für Ein-Modul-Programme.

10.2.7 Compileranweisungen für Objektdateien

10.2.7.1 Interne Übersetzung

Zwischen dem Compileranweisungs-Requester der Projektverwaltung und dem globalen Compiler-Dialogfenster besteht auch optisch eine gewisse Ähnlichkeit. Betrachen Sie einmal die Gadgets, die benutzbar werden, wenn für eine Objektdatei der Modus ***„Intern compilieren"*** angewählt wurde. Sie werden feststellen, daß die wichtigsten Compileroptionen für jedes Modul des Projekts separat eingestellt werden können.



Allgemein gelten die globalen Einstellungen auch im Make-Modus, sofern eine Option nicht explizit anders eingestellt wird. Deshalb finden Sie im Compileranweisungs-Requester eine ganze Menge kleiner Cycle-Gadgets mit den Stellungen ***„Def"***, ***„Ein"*** und ***„Aus"***. Dabei steht ***„Def"***, wie man sich denken kann, für „Default", d. h. bei dieser Schalterstellung soll für das Modul die globale Einstellung gültig sein.

Bei den mit „Optionen" überschriebenen Gadgets entsprechen die ersten sieben exakt den entsprechenden globalen Schaltern. ***„Auto-Make"*** entspricht einem Gadget aus dem „System"-Requester und legt fest, ob die Abhängigkeiten der betreffenden Objektdatei vom Make automatisch gesetzt werden sollen.

Natürlich fehlen noch ein paar Compileroptionen, z. B. ist es über die Requester nicht möglich, für eine Übersetzungseinheit die Include-Pfade anders zu setzen. Solche (wohl eher ungewöhnliche) Einstellungen sind aber möglich, wenn Sie die Projektdatei in den Editor laden, „von Hand" verändern, abspeichern und dann erneut als Makedatei laden. In Abschnitt 3.3 erfahren Sie, wie die Projektdateien aufgebaut sind.

10.3 Handgemachte Makedateien

10.3.1 Die Grundlagen

Das Format der Projektdateien von MaxonPASCAL orientiert sich an den Standard-Makefiles, ist aber natürlich etwas strikter, da in der integrierten Entwicklungsumgebung ja nur ein gewisser Ausschnitt des bekannten Make implementiert werden konnte. Es ist dadurch relativ leicht, ein solches Makefile von Hand zu erstellen oder ein vorhandenes Makefile an MaxonPASCAL anzupassen. Wenn Sie nicht in der integrierten MPASCAL-Umgebung arbeiten, sondern die Option „-m" des mpc benutzen, bleibt Ihnen sogar nichts anderes übrig, als Make-Dateien grundsätzlich manuell zu erstellen.

Es gibt zwei verschiedene Arten von Zeilen, nämlich Abhängigkeits- und Anweisungszzeilen. Sie unterschieden sich formal dadurch, daß letztere mit einem Leerzeichen oder Tabulator beginnen.

Ein „Eintrag" beginnt grundsätzlich mit einer Abhängigkeitszeile, die dem Make sagt, was erzeugt werden kann und von welchen Dateien es abhängig ist. Optional können eine oder mehrere Anweisungszeilen folgen, die festlegen, wie das fragliche Objekt erzeugt werden muß.

10.3.2 Der Linkeraufruf

MaxonPASCAL erwartet als ersten Eintrag stets die Exe-Datei, um die es geht (Sie erinnern sich: Leider immer nur ein Exefile pro Makefile). Die Exe-Datei ist - formal gesehen - von den Objektdateien abhängig, und erzeugt wird sie durch einen Linker-Aufruf. Als Linker muß hier der im mpc benutzt werden (auch wenn Sie nicht mpc, sondern MPASCAL verwenden). In diesem Zusammenhang sollte ich wohl erwähnen, daß es für das Verständnis dieses Kapitels enorm hilfreich wäre, wenn Sie schon ein wenig mit dem mpc vertraut wären.

Eine Abhängigkeitszeile beginnt stets mit dem Namen der zu erzeugenden Datei, gefolgt von einem „:" und optional der Liste der Abhängigkeiten. Wenn unser Projekt „Demo" heißt und aus den beiden Modulen „main.o" und „test.o" besteht, beginnt unser Makefile also mit den Zeilen:

```
Demo: main.o test.o
 mpc main.o test.o -o Demo
```

Dazu noch einige Anmerkungen:

- Da das MaxonPASCAL Make keinen externen Linker aufruft, sondern alles implizit intern erledigt, ist auch die Option „-o Demo", die den Namen der Exe-Datei festlegt, strenggenommen nicht erforderlich, zumal er aus der Abhängigkeitszeile bereits bekannt ist. Aus Ästhetischen Gründen sollte man diese Angabe aber trotzdem machen.
- MaxonPASCAL Make weiß, das ein Linkeraufruf von den benutzten Objektdateien abhängig ist. Deshalb sind auch die beiden Einträge in der Abhängigkeitszeile nicht wirklich nötig.

In der Linkeranweisung sind auch die benutzten Units mit der in Abschnitt 2.5.6.6 Option „-U" aufzuführen, da der Linker ja nicht wissen kann, welche Units für das Projekt benötigt werden. Dies entspricht den Textgadget „Make-Units" im Compiler-Requester.

Beispiel:

```
pc main.o test.o -o test -U printer -U windows/pascalcrt
```

10.3.3 Internes Compilieren

Nach dem Linkeraufruf folgen in beliebiger Reihenfolge die Einträge für die einzelnen Objektdateien. Wenn Ihnen danach ist, können Sie auch Einträge für Projektdateien hinzufügen, die überhaupt nicht zum Projekt gehören - die werden dann ignoriert.

Auch hier beginnt jeder Eintrag mit einer Abhängigkeitszeile, also z. B. main.o: main.p test.h

Die Objektdatei „main.o" soll also bitteschön übersetzt werden, wenn eine der Dateien „main.p" und „test.h" in letzter Zeit verändert wurde. Fehlt noch der Compileraufruf: Der Compiler heißt ja bekanntlich genau wie der Linker „mpc" (oder auch „cp", siehe oben). Die Option „-c" sorgt dafür, daß eine Objektdatei statt einer Exe-Datei erzeugt wird:

```
main.o: main.p test.h
        mpc -c main.p
```

Außer „-c" kann auch jede andere Compiler-Option aus Kapitel 2.5.6.6 benutzt werden, z. B. „-gB" zum Einschalten der Abbruchmöglichkeit. Bei allen Optionen, die nicht explizit gesetzt werden, werden die globalen Defaults verwendet.

Es ist übrigens nicht zwingend erforderlich, dem Make zu sagen, daß ein Objekt von seiner Quelltextdatei (und das gesamte Projekt von seinen Objektdateien) abhängig ist, denn das weiß Make auch so. Also könnte man das Makefile folgendermaßen abkürzen:

```
Demo:  mpc main.o test.o
test.o:
        mpc -c test.p
main.o: test.h
        mpc -c main.p
```

10.3.4 Externe Übersetzung

Jede Compileranweisung, die mit „mpc" beginnt, wird von MaxonPASCAL „intern" ausgeführt, d. h. Make und Compiler sind Bestandteile eines einzigen Programms und keines von beiden muß irgendwie nachgeladen werden. Leider ist MaxonPASCAL zugegebenermaßen nicht ausreichend (z. B. hat man öfters ein Assembler-Modul in seinem Projekt). Aber auch das ist kein Problem: Schreiben Sie die Anweisung ganz einfach in das Makefile! Sie können dann sogar mehrere nacheinender auszuführende Zeilen (=Shell-Anweisungen) für eine Objektdatei angeben, z. B. so:

```
sub.o: sub.asm
       mp:bin/masm sub.asm -c clos     echo "Assembler ist
fertig"
```

Wenn Sie diese Projektdatei von der integrierten Umgebung MPASCAL ausführen lassen, wird vorher sogar geprüft, ob die Dateien, von denen das Objekt abhängig ist (in diesem Falle nur „sub.asm"), sich im Editor befinden und seit ihrer letzten Abspeicherung verändert wurden. Falls ja, werden sie automatisch abgespeichert, denn der externe Compiler/Assembler liest seinen Quelltext ja aus der Datei und nicht (wie MaxonPASCAL) direkt aus dem Edward.

Speziell bei der Benutzung des Maxon Assemblers gibt es noch eine Besonderheit: Nach der Übersetzung schaut MPASCAL nach, ob eine zur Objektdatei passende „.err"-Datei erzeugt wurde (im Beispiel also „sub.err"). Falls ja, versucht MPASCAL diese als MaxonAsm-Fehlerdatei zu interpretieren und die Fehlermeldungen direkt im Compilerfenster anzuzeigen. Also wäre hier der Aufruf masm sub.asm -c clos -v sub.err höchst empfehlenwert.

Noch einmal zurück zu den Mehrfach-Anweisungen: Wie gibt man die eigentlich in MPASCAL ein, da es doch im Compileranweisungs-Requester für externe Übersetzeraufrufe nur ein Textgadget gibt? Nun, hier sind die Zeilen mit einem Gegenschrägstrich „\" zu trennen. Davor und dahinter muß aus Gründen der Eindeutigkeit jeweils ein Leerzeichen stehen.

10.3.5 Weitere Features

Bisher wurden die Grundfunktionen von Make erläutert, die vom internen Make direkt und sichtbar unterstützt werden. Es gibt aber noch eine ganze Menge zusätzlicher Features, die besonders dann von Interesse sind, wenn man seine Projektdatei für den CLI-Compiler mcppc von Hand schreibt.

10.3.5.1 Zeilenverbindung

Die Anzahl der Dateien in einem Projekt wächst oft schnell an und mit ihr auch die Längen von Abhängigkeitszeilen oder Linkeraufrufen. Da z. B. Edward die Zeilenlängen auf 256 Zeichen beschränkt, stellt sich irgendwann das Problem, daß man mehr in eine Zeile packen möchte als eigentlich gesund ist.

Die Lösung ist die Zeilenverbindung mit dem Gegenschrägstrich. Endet eine Zeile mit „\", so verbindet Make sie mit der nächsten Zeile. Auf diese Weise können beliebig viele „echte" Zeilen zu einer einzigen Make-Zeile zusammengefaßt werden.

10.3.5.2 Kommentare

Kommentare beginnen für Make mit einem „#".

Beispiel:

```
# makefile für "Demo"
# Version vom 24.05.94
Demo: main.o # heute nur eine Objektdatei
mpc main.o -o Demo    # Linkeranweisung
```

Da Kommentare dem Make selbst ziemlich egal sind, werden sie auch nicht irgendwie gemerkt. Aber immerhin merkt sich MaxonPASCAL Make beim Lesen einer Projektdatei, ob darin Kommentare, Makros oder Regeln (siehe dazu die nächsten Abschnitte) vorkamen. Haben Sie als Benutzer von MPASCAL in der integrierten Umgebung an der Projekdatei etwas verändert, so daß das Makefile neu geschrieben werden muß, dann fragt MPASCAL sicherheitshalber noch einmal nach, ob das offensichtlich von Hand erstellte Makefile wirklich überschrieben werden soll. Es erscheint also eine Dialogbox mit dem Inhalt

Die Projektdatei „<name>" wurde nicht von MaxonPASCAL Make automatisch erstellt.

Wollen Sie sie trotzdem überschreiben?

Ja **Nein**

Sie haben dadurch bei Bedarf noch Zeit, z. B. eine Shell zu öffnen und Ihr mühsam handgeschriebenes Makefile vor der Vernichtung durch MPASCAL zu retten. Sie können die obige Nachfrage natürlich auch verneinen, aber dann wird das Update der Projektdatei nur aufgeschoben, nicht aufgehoben, und bei der nächsten Gelegenheit nervt MPASCAL erneut mit der selben Frage.

10.3.5.3 Makros

Auch einfache Makros können in Make verwendet werden. Dieses Feature geht aber keineswegs so weit wie z. B. die Makros im PASCAL-Preprozessor, vielmehr kann man im Text lediglich eine Zeichenfolge (den Makronamen) durch eine andere ersetzen lassen. Aber auch das ist hin und wieder ganz nützlich.

Man definiert sich ein Makro durch eine Zeile der Form

<Makroname> = <Inhalt>

Als Makroname ist dabei eine beliebige Folge von Buchstaben, Ziffern und dem Unterstrich „_" erlaubt.

```
Objects = main.o test.o sub.o
link_2 = mpc -g40 -gsi -l amiga
X = Exefile
```

Nachdem ein Makro auf diese Weise definiert wurde, kann es benutzt werden. Dazu muß dort, wo das Makro durch seinen zugewiesenen Wert ersetzt werden soll, ein Dollarzeichen „$" vorangestellt werden. Ist der Makroname mehr als ein Zeichen lang, muß er außerdem auch noch in runden Klammern eingeschlossen werden. Nach den obigen Definitionen könnte eine Make-Datei also so beginnen:

```
$X: $(Objects)
$(link_2)
$(Objects)
```

Daran sehen Sie auch schon die beiden typischen Anwendungen von Makros im Makefiles: Zum einen spart man bei geeigneter Anwendung Tipparbeit, zum anderen muß man für bestimmte Änderungen wie das Hinzufügen weiterer Objektdateien nur noch an einer Stelle der Datei die Makrodefinition verändern, statt an mehreren Stellen den gleichen Mist einzutippen. Nebenbei räumen geschickt gewählte Makros den Quelltext auf und sorgen so für mehr Übersicht.

Es gibt einige vordefinierte Spezialmakros:

„$$" ist fest mit dem Dollarzeichen belegt. Wenn Sie an irgend einer Stelle Ihrer Projektdatei ein einzelnes „$" haben sollen, z. B. als Bestandteil eines eigentümlichen Dateinamens wie „a$b", müssen Sie also ein doppeltes Dollarzeichen schreiben („a$$b"), um Make davon abzuhalten, das nachfolgende (hier also „b") als Makro zu behandeln. Außerdem gibt es noch „$*" und „$@", die bei Regeln eine wichtige Rolle spielen:

10.3.5.4 Regeln

Es ist auf die Dauer ziemlich ermüdend, immer wieder von Hand einzutippen, daß Make aus einer auf „.asm" endenden Textdatei eine „.o"-Objektdatei erzeugen soll, indem es einem bestimmten Assembler mit bestimmten Optionen aufruft. Dieser Vorgang kann durch Make-Regeln erheblich abgekürzt werden.

Die Definition einer Regel sieht auf den ersten Blick wie ein ganz normaler Makefile-Eintrag aus, nur daß der Name der (Pseudo-)Zieldatei aus den Endungen von Quell- und Zieldatei zusammengesetzt ist und keine Abhängigkeiten angegeben werden dürfen. Eine typische Regel für Assembler-Module könnte z. B. so aussehen:

```
.asm.o:     masm -c clos $*.asm -o $@
```

Das Makro „$*" wird durch den seiner Endung beraubten Dateinamen, „$@" durch den vollständigen Namen der Objektdatei ersetzt.

MaxonPASCAL Make wendet Regeln nur an, um Objektdateien zu erzeugen, die in der Abhängigkeitszeile der Linkeranweisung genannt werden und für die es keinen Eintrag mit einer expliziten Compileranweisung gibt. Dann wird nach der ersten Regel gesucht, bei der die Endung der Zieldatei (meist „.o") stimmt und eine entsprechende Quelldatei auch existiert.

Beispiel:

```
.p.o:  pc -c -pc -O $*.p
Demo:  main.o test.o
       pc main.o test.o sub.o -o Demo
main.o:       main.p test.h
```

Wenn nun die Dateien „main.p", „test.p" und „sub.p" existieren, wird für „test.o" die erste, für „main.o" die zweite und für „sub.o" überhaupt keine Regel benutzt - denn wenn „sub.o" nicht in der Abhängigkeitszeile der Exe-Datei auftaucht, sieht Make auch keinen Handlungsbedarf zur Erzeugung derselben.

Für „main.o" wurde im Beispiel sogar eine Abhängigkeitszeile eingeführt. Gibt es eine solche nicht, setzt Make automatisch den Namen der Quelldatei als einzige Abhängigkeit.

Es gibt beim MaxonPASCAL-Make zwei eingebaute Regeln:

```
.p.o:     pc -c $*.p übersetzt Pascal-Quelltexte.
```

Beachten Sie, daß MaxonPASCAL Make bei interner Übersetzung als Compiler-Optionen die global eingestellten Defaults verwendet. Außerdem gibt es eine „.asm.o"-Anweisung für Assembler-Module. In der integrierten Umgebung MPASCAL kann die zugehörige Anweisungszeile im System-Dialogfenster eingegeben werden (übrigens die selbe Anweisung, die auch benutzt wird, wenn der Menüpunkt **„Text und Objekt aufnehmen"** auf eine „.asm"-Datei angewendet wird).

11. Standard-Units

Auf Ihren MaxonPASCAL-Disketten finden Sie einige fertige Units, die Sie in Ihren Programmen verwenden können. Diese Unit-Sammlung wird ständig erweitert (insbesondere sind Sie, jawohl, genau Sie (!), aufgerufen, von Ihnen geschriebene nützliche Units der Allgemeinheit zur Verfügung zu stellen). Deshalb können wir schlecht alle mitgelieferten Units in diesem Handbuch dokumentieren. Also folgt jetzt eine Beschreibung der wichtigsten Standard-Units, zu den übrigen werden Sie Dokumentationen als ASCII-Datei auf der Source-Diskette finden.

11.1 Crt und PASCALCrt

Das Unit Crt öffnet in seinem Initialisierungsteil auf der Workbench ein Window geeigneter Größe sowie ein Console-Device dazu und verbiegt mit *„SetStdIO"* die Standard-Ein-/Ausgabe auf dieses Fenster. Also erspart dieses Unit Ihnen viel Routinearbeit, wenn Sie für die Ein- und Ausgaben eines Programms ein eigenes Fenster öffnen wollen.

Außerdem stellt Crt einige Prozeduren und Funktionen zur Verfügung, die Sie vielleicht schon vermißt haben: Mit *„ReadKey"* und *„KeyPressed"* ist es endlich möglich, einen Tastendruck abzufragen (beim AMIGA ist dies nicht ohne weiteres über die Standard-EA, die ja ein „CON:"-Window ist oder wenigstens so tut als ob, möglich). Durch die Funktion „WindowClosed" können Sie das Betätigen des Fensterschließsymbols abfragen. Auch andere Ereignisse können Sie bequem abfragen, und da Sie Rastport und Userport des Fensters kennen, können Sie im Ausgabefenster sogar (zusammen mit den Units „Intuition" und „Graphics") Grafik ausgeben oder Gadgets und Menüs verwenden.

Auch wenn Sie alle diese Features nicht brauchen, können Sie das Unit „Crt" benutzen, um Ihre Programme von der Workbench startbar zu machen: „Uses Crt" bewirkt, daß beim Programmstart automatisch ein Window geöffnet und die Standard-EA darauf umgeleitet wird - sonst laufen die Programme genau wie zuvor.

Während „Crt" in Assembler programmiert wurde, um den Code möglichst kompakt zu halten, ist „PASCALCrt" (im Unterverzeichnis „Windows") eine sonst vollkommen identische Implementation des Units „Crt" in PASCAL. Dieses Unit liegt Ihnen auch als Quelltext vor, so daß Sie es bei Bedarf modifizieren können.

Es ist aber nicht sinnvoll, in einem Programm sowohl „Crt" als auch „PASCALCrt" zu benutzen, denn dann werden ja gleich zwei identische Ausgabefenster geöffnet.

Folgende Prozeduren und Funktionen werden jeweils in den beiden Units definiert:

Function KeyPressed: Boolean

Die Funktion KeyPressed wird „true", wenn eine Taste gedrückt wird bzw. sich bereits im Tastaturpuffer befindet oder am UserPort des Ausgabefensters eine Nachricht eintrifft bzw. schon anliegt.

Beispiel:

```
Repeat
  Write('Hallo! ')
  Until KeyPressed;
```

Das Programmfragment gibt so lange ununterbrochen „Hallo" aus, bis eine Taste gedrückt oder das Fensterrschließsymbol angeklickt wird.

Procedure WaitForKey

Die Prozedur wartet, bis eine Taste gedrückt wird oder ein Ereignis eintritt. Falls sich bereits eine Taste im Puffer befindet oder eine Nachricht am Window-UserPort anliegt, wird natürlich nicht gewartet.

WaitForKey verbraucht beim Warten keine Rechenzeit, ist also solch' fragwürdigen Konstrukten („busy wait") wie

```
Repeat Until KeyPressed
```

in jedem Fall vorzuziehen.

Function ReadKey: Char

Diese Funktion liest ein Zeichen von der Tastatur. Dabei wird zuvor durch Aufruf von *„WaitForKey"* auf eine Taste oder ein Ereignis (Message) gewartet. Falls keine Taste gedrückt, statt dessen aber eine Message empfangen wurde, gibt ReadKey den Wert chr(0) zurück.

Beispiel:

```
Uses Crt;
Var c: Char;
Begin
  Repeat
 c:= ReadKey;
    WriteLn('Ascii-Code', ord(c):4)
  Until c = chr(13)
End.
```

Das Programm holt Zeichen von der Tastatur und gibt deren ASCII-Code aus, bis der Code der Return-Taste gelesen wird. ReadKey bewirkt kein Echo, d. h. ein gelesenes Zeichen wird nicht automatisch auf den Bildschirm ausgegeben.

Function MessageReceived (Maske): Boolean

Maske: LongInt

Wenn *„KeyPressed"* wahr wird, kann das zwei Ursachen haben: Entweder wurde eine Taste gedrückt oder am UserPort des Ausgabefensters wurde eine Nachricht empfangen. Diese Messages enthalten

Informationen über verschiedene Ereignisse, z. B. Betätigung des Fensterschließgadgets, Anwahl eines Pull-Down-Menüs usw.

Falls eine derartige Nachricht anliegt, liefert *„ReadKey"* den Wert Chr(0), und man kann mit *„MessageReceived"* bestimmen, um welche Art von Nachricht es sich handelte.

Im Interfaceteil des Units „Crt" werden die folgenden Konstanten definiert:

```
NEWSIZE          = $00000002; { Änderung der Fenstergröße }
REFRESHWINDOW    = $00000004; { Fensterinhalt neu aufzubauen }
MOUSEBUTTONS     = $00000008; { Drücken/Loslassen einer Maustaste }
MOUSEMOVE        = $00000010; { Mausbewegung }
GADGETDOWN       = $00000020; { Gadget angeklickt }
GADGETUP         = $00000040; { Gadget losgelassen }
REQSET           = $00000080;
MENUPICK         = $00000100; { Menü angewählt }
_CLOSEWINDOW     = $00000200; { Close-Gadget }
RAWKEY           = $00000400; { Tastendruck, Key-Code }
REQVERIFY        = $00000800;
REQCLEAR         = $00001000;
MENUVERIFY       = $00002000;
NEWPREFS         = $00004000; { Ändern der Preferences }
DISKINSERTED     = $00008000; { Einlegen einer Diekette }
DISKREMOVED      = $00010000; { Herausnehmen einer Diskette }
WBENCHMESSAGE    = $00020000;
ACTIVEWINDOW     = $00040000; { Fenster aktiviert }
NACTIVEWINDOW    = $00080000; { Fenster deaktiviert }
DELTAMOVE        = $00100000; { Verschiebung bei Gadgets }
VANILLAKEY       = $00200000; { Tastendruck, ASCII-Code }
INTUITICKS       = $00400000;
```

Diese Zahlen heißen „IDCMP-Flags" („IDCMP" = „Intuition Direct Communication Message Port").

Man kann mit

```
If MessageReceived(NEWSIZE) Then ...
```

abfragen, ob die Fenstergröße verändert wurde. Da aber jedes Signal einem Bit entspricht, kann man auch abfragen, ob eins von mehreren Signalen empfangen wurde, indem man die Codes mit OR verknüpft:

```
Repeat
  ...
Until MessageReceived(DISKREMOVED Or _CLOSEWINDOW);
```

Diese Schleife wird ausgeführt, bis eine Diskette aus dem Laufwerk entnommen oder das Fenster geschlossen wurde.

...

Vorher muß man aber dem Betriebssystem mitteilen, von welchen Ereignissen man überhaupt erfahren möchte, und zwar mit den Prozeduren „*SetIDCMP*“ und „*AddIDCMP*“. Als Default ist beim Öffnen des Fensters nur das „*_CLOSEWINDOW*“-Flag gesetzt. „*MessageReceived*“ holt die Nachricht aber nicht ab - Sie müssen mit der „*ReadKey*“-Funktion das Chr(0)-Zeichen lesen, damit das Unit „Crt“ weiß, daß Sie die Message ausgewertet haben.

Eine Auswertung der Fenster- und/oder Tastaturereignisse könnte also so aussehen:

```
Uses Crt;
Var c: Char;
Begin
   AddIDCMP (MOUSEBUTTONS); { zusätzlich Mausklicks abfragen }
   Repeat
     c:= ReadKey;
     If c = chr(0) Then
       Begin
         { Nachricht auswerten: }
         If MessageReceived(MOUSEBUTTONS) Then
           Writeln(' Klick!');
         If MessageReceived(_CLOSEWINDOW) Then
           Writeln(' Bye bye!');
         { usw. }
       End
     Else
       Begin
        { Taste auswerten, z. B. so: }
         Write(c);
      End;
   { Programmende mit Close-Gadget oder Esc-Taste: }
   Until (MessageReceived(_CLOSEWINDOW)) Or (c=#x);

End.
```

Natürlich sind beide Teile optional, d. h. wenn Sie nur Tasten oder nur Messages auswerten wollen, können Sie die jeweils anderen Ereignisse ignorieren. Wichtig ist nur, daß Sie auch zur Message-Behandlung die Funktion „*ReadKey*“ benutzen müssen, denn sonst kann das Unt „Crt“ ja nicht wissen, wann Sie eine Nachricht ausgewertet haben und die nächste haben wollen.

Procedure SetIDCMP (Maske)

Maske: LongInt

Diese Prozedur definiert die IDCMP-Flags des Ausgabewindows (siehe unter „*MessageReceived*“). Als Default ist am Anfang nur „*_CLOSEWINDOW*“ gesetzt.

Procedure AddIDCMP (Maske)

Maske: LongInt

Im Gegensatz zu „*SetIDCMP*", das die IDCMP-Flags definiert, fügt die Prozedur „*AddIDCMP*" nur neue Flags hinzu. Der Parameter wird also mit den bisherigen Flags OR-verknüpft. Somit ist „AddIDCMP(0)" eine leere (nichts bewirkende) Anweisung, während „SetIDCMP(0)" alle Flags löschen würde.

Procedure SubIDCMP (Maske)

Maske: LongInt

Dies ist das Gegenstück zu „*AddIDCMP*": Die als Parameter übergebenen Flags werden aus der IDCMP-Maske des Fensters gelöscht.

Function WindowClosed: Boolean

Diese Funktion ist eine Abkürzung für „MessageReceived(_CLOSEWINDOW)".

Man kann damit bequem abfragen, ob das Close-Gadget des Fensters angeklickt wurde. Es gilt aber dasselbe wie bei „*MessageReceived*": Die Ereignisse müssen mit der „*ReadKey*"-Funktion abgeholt werden, damit das „Crt"-Unit weiß, daß sie bearbeitet wurden.

Beispiel:

```
Uses Crt;
 Var c: Char;
 Begin
   Repeat
     c:= ReadKey
   Until WindowClosed
 End.
```

Ohne den „ReadKey"-Aufruf wäre dies eine Endlosschleife, denn die entsprechende Message würde nie am Userport des Windows abgeholt; außerdem wäre es ein „Busy Wait", der Rechenzeit verschwendet.

Function CrtMessage: Ptr

Falls ein Aufruf von „*MessageReceived*" ergibt, daß eine Nachricht empfangen wurde, liefert diese Funktion einen Zeiger auf dieselbe (sonst „NIL"). Man kann sie dann näher auswerten.

Beispiel:

```
Uses Crt, Intuition;
 Var c: Char;
     m: ^IntuiMessage; { im Unit "ntuition" definiert }
 Begin
   AddIDCMP(MOUSEBUTTONS);
   Repeat
     c:= ReadKey;
     If MessageReceived(MOUSEBUTTONS) Then
```

```
      Begin
        m:= CrtMessage;
        Writeln('Position:', m^.MouseX:5, m^.MouseY:5);
      End
  Until WindowClosed
End.
```

Wenn Sie mit der linken Maustaste in das Ausgabefenster klicken, gibt dieses Programm die genauen Koordinaten der Position an.

Function CrtWindow: Ptr

Diese Funktion liefert die Windowhandle des Crt-Windows. Sie brauchen Sie für viele Zwecke, z. B. wenn Sie das Fenster mit einem Gadget oder Menü versehen wollen.

Function CrtRastPort: Ptr

Mit dieser Funktion erhalten Sie einen Zeiger auf den RastPort des Crt-Fensters. Diesen Pointer benötigt man vor allem für Grafikbefehle.

Beispiel:

```
Uses Crt, Graphics;
Var Eingabe: Char;
    i: integer;
    x1, x2, y1, y2: Real;
Begin
  Write(#e'0 p');
          { Cursor unsichtbar machen }
  SetAPen(CrtRastPort, 1);
  { Zeichenfarbe: weiß }
  For i:=1 to 200 Do
    Begin
      x1:= 320+300*sin(i/25);
      x2:= 320-300*sin(i/20);
      y1:= 120+100*cos(i/32);
      y2:= 120+100*cos(i/40);
      Move(CrtRastPort, Round(x1), Round(y1));
      Draw(CrtRastPort, Round(x2), Round(y2))
    End;
  Repeat Eingabe:= ReadKey Until WindowClosed
End.
```

Dieses Programm gibt mit Hilfe dreier Funktionen der Graphics-Library (die in Unit „Graphics" definiert wird) ein recht hübsches Muster aus.

Im entsprechenden Abschnitt der AMIGA-Einführung in diesem Buch finden Sie Näheres über die Grafikfunktionen.

Function CrtUserPort: Ptr

Gibt einen Zeiger auf den UserPort des Crt-Fensters.

Function NameOfProgram: Str

Diese Funktion gibt den Namen des Crt-Fensters aus. Im allgemeinen wird hier der Name genommen, unter dem das Programm als Exe-File von der Workbench oder vom CLI geladen wurde. Falls es aber aus der MaxonPASCAL-Entwicklungsumgebung gestartet wurde, gibt es zwei Sonderfälle: wurde MaxonPASCAL aus dem CLI gestartet, erhält das Console-Fenster die entsprechende CLI-Anweisung, beim Workbenchstart wird „MaxonPASCAL" als Default benutzt.

Function CrtConsole: Ptr

Von dieser Funktion erhalten Sie die Handle des Console-Devices, das vom Unit „Crt" für das Ein-/Ausgabefenster geöffnet wird. Wenn Sie in Ihrem Programm ein zweites Window öffnen und mit „*SetStdIO*" die Standard-EA vorübergehend darauf umleiten, können Sie sie nachher wieder mit „SetStdIO(CrtConsole)" auf das normale Crt-Fenster zurücksetzen.

Function TextLines: integer

Diese Funktion gibt an, wieviele Textzeilen im Ausgabefenster Platz haben.

Function TextColms: integer

Gegenstück zu „*TextLines*": Gibt die Anazhl der Textspalten des Windows zurück.

Beispiel:

```
USES Crt;
BEGIN
  Writeln('Das Fenster ist ',TextColms,'*',TextLines,' Zeichen
     groß.');
  WaitForKey
End.
```

Function WhereX: integer

Mit dieser Funktion können Sie die aktuelle Cursorposition erfragen. Sie gibt die Spalte zurück, in der der Cursor steht.

Function WhereY: integer

Liefert analog zu „WhereX" die Spalte, in der der Cursor gerade steht.

Procedure WindowTitles (windowname, screenname: Str)

Das Unit „Crt" gibt dem Window den Namen, unter dem das Programm gestartet wurde. Mit der Prozedur *„WindowTitles"* kann dieser Name sowie der Name der Workbench geändert werden. Der Titel der Workbench ändert sich aber nur solange, wie das Crt-Fenster aktiv ist.

Wenn Sie einen Parameter auf „Nil" setzen, entspricht dies keinem Namen. Der Wert „Str(-1)" bewirkt, daß der bisherige Name erhalten bleibt.

Beispiele:

```
WindowTitles ('Test', Str(-1))
```

Window heißt „Test", Name des Workbenchscreens bleibt erhalten

```
WindowTitles (Dateiname, 'Editor');
```

Der Fenstername entspricht dem Inhalt der Variablen „Dateiname" (z. B. eine Stringvariable), während der Workbenchscreen bei aktivem Crt-Fenster den Namen „Editor" erhält. Achtung: Wenn Sie den Inhalt der Stringvariablen ändern, wird der Fenstername NICHT automatisch aktualisiert (woher soll das Betriebssystem das auch wissen), sondern erst beim nächsten Refresh der Titelzeile. Sie sollten also bei jeder Zuweisung an die Variable erneut die *„WindowTitles"*-Prozedur aufrufen.

```
WindowTitles (Str(-1), Nil);
```

Der Fenstername bleibt erhalten, die Workbench wird namenlos.

11.2 Printer

Dieses Unit ist in PASCAL geschrieben und ausgesprochen kurz: Im Interfaceteil wird lediglich eine Text-Dateivariable namens „Lst" definiert und im Init-Teil unter dem Namen „PRT:" zum Schreiben geöffnet. Falls dies nicht gelingt, bricht das Programm mit einer entsprechenden Fehlermeldung ab.

Durch Einbinden dieses Units können Sie also einfach mit

```
Write (Lst, ...)
```

Daten auf den Drucker ausgeben.

11.3 Die AMIGA-System-Units

Zum professionellen Programmieren sind auf dem AMIGA die System- Includedateien unverzichtbar. Sie enthalten zahllose Deklarationen von Konstanten, Datentypen und Library-Funktionen und ermöglichen so den vollen Zugriff auf das AMIGA-Betriebssystem.

Das Problem bei diesen Includefiles ist, daß es ganz schön viele werden können und das Übersetzen somit spürbar Zeit kostet. Um dies etwas zu beschleunigen, wurden zu den wichtigsten Includefiles Units erstellt. Ihr Interfaceteil besteht im wesentlichen aus den Includefiles, während ihr

Implementationsteil meist leer ist (bei einigen werden im Init-Teil kleine Initialisierungen vorgenommen).

Wegen des enormen Umfangs dieser Includefiles kann an dieser Stelle nicht jeder dort definierte Bezeichner erwähnt (und schon gar nicht erläutert) werden. Einiges wird in diesem Handbuch noch im Kapitel über Systemprogrammierung erklärt, sonst kann ich Ihnen nur empfehlen, sich die entsprechenden Includefiles einmal anzusehen und auf die diverse Literatur zur AMIGA-Systemprogrammierung verweisen.

11.3.1 Exec1 und Exec

Exec ist der innerste Teil des AMIGA-Betriebssystems. Deshalb werden einige der Exex-Includes auch von anderen Betriebssystemteilen benutzt, während man andere nur braucht, wenn man wirklich direkt Exex programmieren will.

Aus diesem Grund wurden zwei Exec-Units definiert: Zunächst „Exec1", das in seinem Interfaceteil folgende Includefiles enthält:

```
exec/lists.h
exec/nodes.h
exec/ports.h
exec/semaphores.h
exec/tasks.h
```

Das Unit Exec1 wird dann auch von so ziemlich allen anderen Systemunits benutzt.

In „Exec" wird dann der ganze Rest definiert, namentlich der Inhalt folgender Include-Dateien:

```
exec/alerts.h
exec/devices.h
exec/execbase.h
exec/errors.h
exec/interrupts.h
exec/io.h
exec/libraries.h
exec/memory.h
exec/resident.h
```

Außerdem wird hier mit „USES" das Unit „Exec1" eingebunden, so daß dann auch die Definitionen dieses Units automatisch zur Verfügung stehen.

11.3.2 GraphTypes und Graphics

Die „Graphics"-Includes sind im wesentlichen in zwei Dateien enthalten: „GraphTypes" und „Graphics". Erstere enthält die wichtigsten Datentypen im Grafik-Bereich, nämlich den Inhalt folgender Dateien:

```
graphics/gfx.unit.h
graphics/clip.h
```

```
graphics/copper.h
graphics/view.h
graphics/layers.h
graphics/rastport.h
graphics/text.h
```

Die Datei „gfx.unit.h" ist identisch mit „gfx.h", enthält aber von der dort deklarierten Funktion „RasSize" nur den Kopf (wie es im Interfaceteil von Units sein muß). Der Rumpf dieser Funktion steht im Implementationsteil des Units. Außerdem benutzt dieses Unit das oben beschriebene Unit „Exec1".

Das Unit „Graphics" benutzt „Exec1" und „Graphtypes". Im Interfaceteil steht nur noch die Includedatei „graphics.unit.lib", die sich von „graphics.lib" darin unterscheidet, daß die Prozedur „OpenGfx" fehlt. Statt dessen wird die Graphics-Library im Initialisierungsteil des Units geöffnet. Wenn Sie das Unit „Graphics" benutzen, müssen Sie diese Bibliothek also weder selbst öffnen noch schließen.

Beachten Sie aber, daß diese beiden Units nicht alle Grafik-Includes repräsentieren. Wenn Sie also z. B. mit Gels arbeiten wollen, müssen Sie die Datei „graphics/gels.h" zusätzlich als Includedatei aufrufen.

11.3.3 Intuition

Im Interfaceteil des Units „Intuition" stehen folgende Includedateien:

```
intuition/intuition.h
intuition/screens.h
intuition.lib
```

Außerdem werden die Units „Exec1" und „Graphtypes" benutzt. Der Inhalt der Dateien „intuition/intuitionbase.h" und „intuition/preferences.h" ist also nicht Bestandteil des Intuition-Units. Im Initialisierungsteil des Units wird die Intuition-Library geöffnet.

11.3.4 Custom

Das Unit „Custom" entspricht der Includedatei „hardware/custom.h" und definiert somit die Struktur der Custom-Chip-Register. Außerdem exportiert das Unit eine Variable:

```
VAR cc: Custom ABSOLUTE $DFF000;
```

Somit ist es möglich, über diese Variable die Chipregister direkt anzusprechen.

Ein extrem einfallsreiches (aber wenigstens kurzes) Beispiel:

```
Uses Custom;
Var i: integer;
Begin
  For i:=0 to MaxInt Do
    cc.color[0]:= i
```

```
End.
```

Dieses kleine Programm ändert für einige Zeit ständig die Hintergrundfarbe.

11.4 ExecSupport und ExecIO

Zwischen der direkten Programmierung der Hardware und dem Zugriff darauf über Libraries gibt es im AMIGA eine weitere Ebene, Geräte wie z. B. den Drucker oder die Diskettenlaufwerke zu programmieren: Die Devices.

Dabei handelt es sich hauptsächlich um Tasks, die ständig laufen (aber natürlich im Waiting State sind, wenn sie gerade nichts zu tun haben, und somit keine Rechenzeit verbrauchen). Andere Tasks können den Devices durch Messages Aufträge erteilen, die dann - falls möglich - vom jeweiligen Device ausgeführt werden.

Ein Device wird von MaxonPASCAL direkt unterstützt: Das Console-Device, das sich um Texteingaben und -ausgaben in Windows kümmert. Außerdem enthält das AMIGA-Betriebssystem standardmäßig folgende Devices:

- Audio Device: erledigt Soundausgaben. Man kann zwar auch über direkten Zugriff auf die Hardware-Register Töne erzeugen, aber über dieses Device geht das „sauberer" und zum Teil auch einfacher.
- Timer Device: kann praktisch nur eine frei wählbare Zeit lang warten und dann melden, daß die Zeit abgelaufen ist. Man kann es brauchen, wenn man in bestimmten Zeitabständen irgendwelche Aktionen ausführen will.
- Trackdisk Device: Damit können Sie direkt auf die Blöcke und Sektoren von Disketten zugreifen. Wenn Sie eine Festplatte besitzen, verfügt Ihr Betriebssystem außerdem über ein Device, das analog zu „Trackdisk" auf die Harddisk zugreift.
- Input Device: handhabt auf unterster Ebene Benutzereingaben aller Art.
- Keyboard Device: Tastaturtreiber - Gameport Device: befaßt sich mit den beiden „Gameports", also den Maus- und Joystickschnittstellen.
- Narrator Device: ist für die (meist grausam klingende und deshalb nur selten benutzte) Sprachausgabe des AMIGA verantwortlich.
- Serial Device: Treiber für die serielle Schnittstelle. Jeder, der schon einmal die entsprechende Schnittstelle unter MS-DOS programmieren mußte, wird von der Leistungsfähigkeit dieses Devices erfreut sein.
- Parallel Device: Treiber für die parallele Schnittstelle.
- Printer Device: befaßt sich mit dem Drucker und kann z. B. auch Grafiken ausdrucken.
- Clipboard Device: unterstützt das Zwischenspeichern und spätere Wiederverwenden von Daten.

Warum ich Ihnen das alles erzähle? Nun, im AMIGA-Betriebssystem „fehlen" einige Routinen, die (hauptsächlich) die Programmierung von Devices komfortabler machen, nämlich die Exec-Support-Funktionen. Sie sind im „AMIGA ROM Kernel Reference Manual: Exec" als Quelltext enthalten - aber,

wie nicht anders zu erwarten, in „C". Diese Funktionen - außer „CreateTask" - stehen Ihnen als MaxonPASCAL-User aber in Form eines Units zur Verfügung: Es heißt „ExecSupport".

Außerdem gibt es noch ein Unit namens „ExecIO", das alles weitere enthält, was man so zum Device-Programmieren braucht.

Aber zunächst zu „ExecSupport". Hier sind im einzelnen folgende Prozeduren und Funktionen implementiert:

PROCEDURE NewList (list: p_List);

Initialisiert die Listenstruktur, auf die der Parameter zeigt.

FUNCTION CreatePort (name: Str; pri: Byte): p_MsgPort;

Erzeugt einen Message-Port unter dem angegebenen Namen und mit der Priorität „pri", und gibt einen Zeiger darauf zurück. Als kleine Besonderheit merkt das Unit ExecSupport sich, welche Ports diese Funktion erzeugt hat, und entfernt diese am Programmende automatisch (mittels eines entsprechenden ExitServers), sofern dies noch nicht geschehen ist.

PROCEDURE DeletePort (port: p_MsgPort);

Ein mit „CreatePort" erzeugter Messageport wird entfernt. Wie bereits oben angedeutet, muß diese Prozedur in Ihrem Programm praktisch nie direkt benutzt werden, da am Programmende automatisch alle erzeugten Messageports gelöscht werden.

FUNCTION CreateStdIO (ioReplyPort: p_MsgPort): Ptr;

Diese Funktion erzeugt eine Standard-IO-Request-Struktur und gibt einem Zeiger darauf zurück. Als Parameter wird ein Zeiger auf einen Messageport benötigt, der z. B. zuvor mit „CreatePort" erzeugt worden sein kann. Wenn die Struktur nicht eingerichtet werden konnte, steigt die Funktion mit einer Fehlermeldung aus. Wie bei „CreatePort" werden auch hier alle eingerichteten IO-Strukturen am Programmende automatisch freigegeben, sofern dies noch nicht geschehen ist.

PROCEDURE DeleteStdIO (ioStdReq: Ptr);

Als Gegenstück zu „CreateStdIO" löscht diese Prozedur eine „normale" IO-Request-Struktur. Allerdings werden diese Strukturen am Programmende auch automatisch gelöscht, so daß Sie „DeleteStdIO" nur selten direkt benutzen werden.

FUNCTION CreateExtIO (ioReplyPort: p_MsgPort; size: Long): Ptr;

Einige Devices benötigen eine um gewisse gerätespezifische Einträge erweiterte IO-Request-Struktur. Mit der Funktion „CreateExtIO" kann man solche Strukturen beliebiger Größe (Parameter

„size" entspricht der Gesamtgröße im Bytes) analog zu „CreateStdIO" erzeugen lassen. Die zusätzlichen Felder werden mit Nullen initialisiert.

Auch diese Funktion steigt bei Mißerfolg mit einer Fehlermeldung aus und merkt sich intern die eingerichteten Strukturen, so daß diese am Programmende automatisch gelöscht werden.

PROCEDURE DeleteExtIO (ioExt: Ptr);

Diese Prozedur löscht eine mit „CreateExtIO" erzeugte Struktur.

PROCEDURE BeginIO (ioReq: Ptr);

Der Aufruf dieser Prozedur entspricht einem Einsprung in eine interne Funktion, die jedes Device besitzt und allgemein eine IO-Operation startet. „BeginIO" ähnelt stark der Exec-Funktion „SendIO", allerdings ist „BeginIO" direkter, d. h. ohne zusätzlichen Exec-"Überbau". Im allgemeinen sollte man „SendIO" benutzen, während „BeginIO" nur in einigen speziellen Fällen sinnvoll benutzt werden kann.

Das Unit „ExecSupport" benutzt übrigens das Unit „Exec1".

Wie bereits erwähnt, gibt es noch ein Unit namens „ExecIO". Es benutzt die Units „Exec1" sowie „ExecSupport" und enthält im Interfaceteil folgende zur Deviceprogrammierung notwendigen Includefiles:

```
'exec/libraries.h'
'exec/devices.h'
'exec/io.h'
```

Außerdem exportiert es noch zwei Prozeduren:

PROCEDURE Open_Device (DevName:Str; Unit:Long; IOReq: Ptr; Flags:Long)

Diese Prozedur entspricht der Exec-Funktion „OpenDevice", bis auf zwei Unterschiede:

1. Konnte das Device nicht geöffnet werden, steigt „Open_Device" mit einer Fehlermeldung aus. Deshalb konnte es auch als Prozedur implementiert werden, während „OpenDevice" eine Function ist, die evtl. eine Fehlernummer zurückgibt.
2. Wieder einmal wird über die geöffneten Devices Buch geführt, so daß sie am Programmende vollautomatisch geschlossen werden und das System immer schön aufgeräumt ist.

PROCEDURE Close_Device (ioRequest: Ptr)

Unterscheidet sich von der Exec-Prozedur „CloseDevice" nur insofern, als daß hier anhand der von „Open_Device" erzeugten Liste geprüft wird, ob das Device auch wirklich geöffnet wurde und daß diese Prozedur verzichtbar ist, da am Programmende alle Devices automatisch geschlossen werden.

A) Compiler-Fehlermeldungen

"," expected.
":" expected.
"(" expected.
")" expected.
"." expected.
".." expected.
"=" expected.
"[" expected.
"]" expected.
":=" expected.
"OF" expected.
"BEGIN" expected.
"END" expected.

Der Compiler vermutet, daß an der angegebenen Stelle das entsprechende Symbol stehen müßte. Dabei ist aber zu beachten, daß er den Fehler nur grob analysiert und im Grunde genommen absolut keine Ahnung hat, was Sie an dieser Stelle eigentlich vorhatten. Deshalb kann der eigentliche Fehler auch an einer ganz anderen Stelle liegen.

Absolute variables must get declared single.

Werden Variablen als „ABSOLUTE" deklariert, müssen sie einzeln deklariert werden. Falsch wäre z. B.

```
VAR a,b: integer ABSOLUTE Adr;
```

richtig dagegen

```
VAR a: integer ABSOLUTE Adr;
    b: integer ABSOLUTE Adr;
```

Boolean Expression expected.

Der Compiler erwartet hier einen Boole'schen Ausdruck, for example als Bedingung bei IF..Then, While..Do oder Repeat..Until.

Branch too far.

Sprünge, die inhärent bei PASCAL-Strukturen wie IF, REPEAT oder CASE auftreten, dürfen im Code nur über eine Strecke von maximal 32767 Bytes reichen. In der Praxis dürfte dieser Fehler kaum auftreten, denn das bedeutet, daß sich z. B. eine Schleifenstruktur über tausende von Programmzeilen

erstreckt. Falls Sie so etwas tatsächlich einmal zustande bringen, sollten Sie den Umfang der Struktur durch Zerlegen in Prozeduren verringern - denn Unterprogrammaufrufe (und auch GOTO-Sprünge) sind von dieser Regelung ausgenommen.

Can not jump into a structure.

Man kann mit GOTO nicht von außen z. B. in eine FOR-Schleife hineinspringen.

Can not open mathieeedoubbas.library!

Vor dem ersten Compiler-Lauf wird die genannte Bibliothek geöffnet. Sie muß sich also im „Libs:"-Verzeichnis befinden.

Can not write unit.

Die Interface-Datei eines Units konnte nicht geschrieben werden.

Can't copy this file.

Bei einem Include-Dateien mit zwei angegebenen Pfaden konnte die Datei zwar über den zweiten Pfad gelesen, nicht aber an den ersten Pfad kopiert werden.

Can't create directories.

Beim Kopieren von Includefiles und Units werden bei Bedarf Unterverzeichnisse automatisch erstellt. Das ging hier schief.

CASE needs an ordinal expression.

Mit CASE können nur geordnete Datentypen verglichen werden, also beispielsweise keine Real- oder String-Werte.

Compiler stack overflow.

Der Task-Stack war nicht groß genug für den Compiler (dürfte kaum vorkommen).

Constant expected.

Tritt an verschiedenen Stellen auf, u. a. im CONST-Definitionsteil und in der CASE-Anweisung. Hier darf nur eine Konstante stehen.

DO expected.

Hinter WHILE, WITH oder FOR fehlt das zugehörige DO. Vielleicht haben Sie sich aber auch in der Bedingung vertan, so daß der Compiler deren Ende an dieser Stelle vorzeitig vermutet.

Double definition

Ein Label wurde zweimal gesetzt.

Double use of Identifier.

Sie haben in einem Deklarationsteil (z.B. Variablendeklaration) einen Bezeichner definiert, der auf selber Ebene schon einmal deklariert ist.

Element type is too large.

Sets dürfen maximal 65536 Elemente enthalten, wobei die Grenzen auf Vielfache von 16 auf- bzw. abgerundet werden.

Error in string

Beim Einschließen von Steuercodes in Stringkonstanten haben Sie irgendetwas falsch gemacht.

Error while reading unit.

Dateifehler beim Lesen des Interface-Teils eines Units

Field identifier expected.

Dies ist kein gültiger Feldbezeichner dieses Recordtyps.

File expected.

Hier hat eine Variable eines Dateityps zu stehen.

File is not of required Type

Die mit „{$link ... }" oder als Unit eingeladene Datei ist keine Objektdatei.

FILE OF FILE is not allowed.

Der Elementtyp eines File-Typs darf weder selbst ein File-Typ sein noch einen solchen enthalten.

File not found

Eine Include-Datei konnte nicht gefunden werden, vielleicht, weil sie gar nicht existiert.

Filename expected.

Hinter der Compiler-Directives „incl" und „path" müssen jeweils Dateinamen stehen, die mit einfachen ' oder doppelten „ Hochkomma einzuschliessen sind.

Filename too long.

Die maximale Länge von Dateinamen für Include-Dateien beträgt einschließlich Pfad 80 Zeichen.

Files must be VAR parameters.

Wenn sie eine Datei als Parameter an eine Prozedur oder Funktion übergeben wollen, muß dies in Form eines VAR-Parameters geschehen.

First constant is higher than second one.

Bei der Definition eines Ausschnittstyps (z. B. eines Array-Indextyps) haben Sie den Bereich so gewählt, daß die obere Grenze kleiner als die untere ist.

FOR needs an ordinal type.

Die Schleifenvariable einer FOR-Schleife muß von einem geordneten Datentyp sein (insbesondere nicht REAL).

Functions can return simple, pointer or string types only.

Der Datentyp, den eine Function zurückgibt, kann sein:

- Ganzzahl, Boolean, Char oder ein Ausschnitt eines solchen Typs
- Real
- Pointertyp
- String

Garbage at end of program.

Hinter dem Programmende steht noch etwas.

Global procedure without parameters expected.

Prozeduren, die als Parameter bei „AddExitServer" angegeben werden, müssen global sein und dürfen keine Parameterliste haben.

Identifier expected.

Der Compiler meint, daß hier ein Bezeichner stehen müßte.

Identifier not defined in that translation unit

Ein Bezeichner wurde falsch qualifiziert, z. B. „System.Hallo", so daß er im entsprechenden Unit nicht gefunden werden konnte.

Illegal type for textfile input

Daten dieses Typs können nicht mit Read/Readln aus einer Textdatei (zum Beispiel von der Tastatur) gelesen werden.

Illegal assignment or compiler failure Type conflict or compiler failure Double panic - general protection!

Hier haben Sie den Compiler ertappt. Es ist intern etwas eingetreten, was eigentlich nicht sein darf. Zu Deutsch: Es handelt sich um einen Bug des Compilers oder um einen fälschlich nicht gemeldeten Fehler Ihres Programms. Wahrscheinlich liegt es daran, daß er sich in der Vielzahl der kombinierbaren unterschiedlichen Datentypen verheddert hat.

Illegal type for textfile output.

Daten dieses Typs können nicht mit Write/Writeln in eine Textdatei (z. B. auf den Bildschirm) geschrieben werden.

IMPLEMENTATION expected.

Im Interfaceteil eines Units trat ein Fehler auf.

Incompatible parameterlists.

Die Parameterliste (oder auch, bei Funktionen, der Ergebnistyp) eines Prozedur/Funktion-Parameters paßt nicht.

Integer expression expected.

An dieser Stelle muß ein ganzzahliger Ausdruck stehen, z. B. als Längenangabe von String[..] oder Feldbreitenangabe bei Write(-Ln).

Label expected.

Hinter dem „LABEL"-Symbol kann entweder ein gewöhnlicher Bezeichner oder eine Ziffernfolge (ganze Zahl) stehen.
oder: Was hier hinter GOTO steht, ist nicht als Label deklariert worden.

Label on wrong level.

Ein Label darf nur auf der Ebene gesetzt werden, auf der es deklariert wurde, insbesondere nicht in einem Unterprogramm dieser Ebene.

Lib-base must be of LongInt or Pointer type.

Die Basisadresse einer Library muß ein Ausdruck eines der genannten Typen sein (vielleicht liegt es daran, daß Sie „IntBase", „DosBase" oder „SysBase" deklariert und dann ein System-Include-Datei geladen haben).

Magic number does not match.

Ein Unit wurde zwar gefunden, aber der Compiler stellte fest, daß es gar kein Unit ist.

Memory overflow

Während des Compilierens ist der Arbeitsspeicher übergelaufen. Verlassen Sie MaxonPascal und versuchen Sie es mit größerem Arbeitsspeicher noch einmal.

Name too long (max. 80 chars)

Dateinamen von Units u. ä. dürfen einschließlich Pfad höchstens 80 Zeichen lang werden.

Nesting overflow

Die Schachtelungstiefe von Prozeduren und Funktionen darf höchstens 16 betragen.

No free memory for buffer.

Aus Geschwindigkeitsgründen wird für Include-Dateien jeweils ein Pufferspeicher angelegt. Dieser Error sagt, daß dafür kein Speicher mehr reserviert werden konnte.

No free memory for BSS-Hunk.

Für die als STATIC oder EXPORT deklarierten Variablen ist kein Speicher mehr frei.

No free memory for relocation table.

Der Compiler muß gelegentlich für die Relokationstabelle (was das ist, braucht Sie nicht unbedingt zu interessieren, falls Sie es nicht wissen) zusätzlichen Speicher reservieren. Wenn Sie diese Meldung erhalten, haben Sie ihr RAM so vollgeknallt, daß nicht mehr einmal ein paar KByte für diese Tabelle frei waren.

No free memory for unit.

Es war kein Speicher zum Einlinken einer Datei frei.

Numeric expression expected.

Hier muß ein numerischer Ausdruck stehen.

Only global Procedures/Functions can get exported.

Lokale Prozeduren und Funktionen dürfen nicht mit „EXPORT" ausgeführt werden.

Ordinal Type expected.
Ordinal expression expected.

Hier dürfen nur geordnete Datentypen stehen, also Boolean, Char, Ganzzahl- und Aufzählungstypen.

Offsets must be integer.

Die Adressoffsets in einer Library müssen natürlich ganzzahlig sein.

Out of range

Zur Compile-Zeit wurde bereits festgestellt, daß irgendwo ein Bereichsüberlauf stattfindet, z. B. bei „chr(257)".

Overflow at nesting of Incl-files

Include-Dateien dürfen maximal 25-fach ineinander verschachtelt werden. Auch die Interface-Teile von Units werden beim Lesen intern als Include-Dateien behandelt (was sie aber nicht sind), so daß dieser Fehler auch hier auftreten kann.

Overflow in floating-point-arithmetic.

Bereits zur Compile-Zeit trat bei Fließkommarechnungen ein Überlauf auf, z. B. durch eine zu große Konstante.

Please set an unit directory.

Wenn Sie Units mit „USES" einbinden oder ein Unit compilieren (wobei es automatisch abgespeichert wird), müssen Sie im Pull-Down-Menü „Suchpfade" mindestens ein Directory dafür angeben. Am besten setzen Sie mit dem PascalPrefs-Programm einen entsprechenden Eintrag in die Config-Datei.

Please use „Round" or „Trunc".

Gleitpunktausdrücke können nicht über normale Typkonvertierungen wie z. B. „Long(x)" oder Integer(Sqrt (a))" in ganze Zahlen gewandelt werden. Dazu sind die genannten Funktionen zu verwenden.

'PROCEDURE' or 'FUNCTION' expected.

Dieser Fehler tritt nur innerhalb einer Library-Deklaration auf.

"Program" expected.

Am Anfang muß nicht unbedingt „Program" stehen. Dieser Fehler wird immer gemeldet, wenn das Programm nicht sinnvoll anfängt.

Short Circuit of Units.

Ein Units ruft sich selbst auf, evtl. auch indirekt über weitere Units.

Simple type expected.

(wird u. a. bei EXCHANGE gemeldet): hier sind nur Variablen einfacher Datentypen erlaubt: Geordnet, Real oder Pointer.

String exceeds line.

Eine Zeichenkette darf das Zeilenende nicht überschreiten. Vielleicht haben Sie bei einem String das Hochkomma am Ende vergessen oder eins gesetzt, wo Sie gar keines setzen wollten.

String expression expected.

Falscher Parameter bei „Insert", „Delete" o. ä.

String too long.

Die maximale Länge von Stringkonstanten beträgt 400 Bytes.

Syntax error in compiler directive.

In einer Compiler-Directive haben Sie irgend etwas falsch gemacht.

Temporary strings can not get assigned to „STR"-pointers.

Die String-Verknüpfung „+" sowie die Funktionen „Concat", „Copy" und „RealStr" liefern einen temporären String zurück, der keinen dauerhaften (also nur „temporären") Speicherplatz besitzt. Deshalb kann man diese Funktionsergebnisse zwar an STRING-Variablen zuweisen, nicht aber an STR-Variablen.

Dieser Fehler wird auch gemeldet, wenn ein temporärer String als Parameter des Typs „Str" übergeben wird.

THEN expected.

Hinter einem IF fehlt das zugehörige THEN. Vielleicht haben Sie sich aber auch in der Bedingung vertan, so daß der Compiler deren Ende an dieser Stelle vorzeitig vermutet.

These types can not be converted.

Eine derartige Typkonvertierung ist nicht möglich.

This function identifier can not be a statement.

Am Anfang eines Befehls - oder dessen, was der Compiler dafür hält - steht unerlaubterweise ein Funktionsname.

This Procedure/Function can not be used as parameter.

Nicht jede Standardfunktion kann als Parameter übergeben werden. Beispiele: chr, sin, cos oder srqt dürfen, ord, eof oder sqr dagegen nicht. Faustregel: Eine Funktion, bei der Parameterliste und Rückgabetyp immer gleich sind, darf übergeben werden („sqrt" bildet immer von Real auf Real ab, „Sqr" aber von jedem numerischen Typ auf denselben, besitzt also mehrere Signaturen). Für Prozeduren gilt natürlich sinngemäß das selbe.

TO expected.

Hinter einem FOR fehlt das zugehörige TO oder DOWNTO. Vielleicht haben Sie sich aber auch im Ausdruck vertan, so daß der Compiler deren Ende an dieser Stelle vorzeitig vermutet.

Type expected.

Hier muß ein Typ stehen, egal, ob als Typbezeichner oder als Beschreibung.

Type conflict.

Zwei Datentypen passen nicht zueinander, z.B. die beiden Konstanten, die die Grenzen eines Ausschnittstyps festlegen.

Type conflict of operands.

Die Typen zweier Operanden passen nicht zusammen.

Type-free pointers must not be used here.

Bei NEW und DISPOSE dürfen Sie keine Variablen des Typs „Ptr" als Parameter angeben, da der Compiler ja nicht weiß, wie viel Speicher er dafür reservieren muß.

Type identifier expected.

Hier darf nur ein Typbezeichner stehen. Typbeschreibungen sind nicht zulässig.

Unable to open main file.

Nach dem Übersetzen eines Units konnte die angegebene Hauptdatei nicht gefunden werden.

Undefined Type.

Dieser Datentyp wurde zwar schon erwähnt, aber noch nicht definiert.

Beispiel: „Type a= ^ b; c=b;" ergibt diesen Fehler.

Unexpected end of source.

Der Quelltext war „überraschend" (für den Compiler) zu Ende.
Mögliche Ursache: Sie haben einen Kommentar begonnen und die Klammer am Ende vergessen.

Unit not found.

Das angegebene Unit wurde nicht gefunden.

Unknown identifier or Syntax error.

Diese Fehlermeldung kann so gut wie alles bedeuten und kann auch an so ziemlich jeder Programmstelle auftreten. Entweder wurde ein Bezeichner nicht deklariert oder ein grober Fehler in der Syntax gefunden.

USES expected.

Hinter „FROM <Projektname>" muß „USES" stehen.

Variable at odd address

Einer als ABSOLUTE deklarierte Variable, die mehr als ein Byte belegt, wurde eine ungerade Adresse zugewiesen. Hier spielt der Prozessor nicht mit.

Variable expected.

Hier soll eine Variable stehen.

WITH expects a variable of a record type.

Hinter WITH muß eine Record-Variable stehen (mit 99%-iger Wahrscheinlichkeit haben Sie an Stelle einer Record-Variablen einen Pointer auf eine solche gesetzt).

Wrong index type.

Sie haben wahrscheinlich beim Array-Zugriff einen Index mit einem falschen Datentyp verwendet.

B) Laufzeitfehler

Subscript out of range.

Der Index eines Arrays (wozu natürlich auch die Strings gehören) war außerhalb des angegebenen Bereichs.

Diese Fehlermeldung erhalten Sie nur, wenn im Menü **„Optionen/Compiler"** der Punkt ***„Indexbereich"*** abgehakt ist.

Out of range.

Einem Ausschnittstyp wurde ein Wert außerhalb des deklarierten Bereichs zugewiesen.

Eine solche Bereichsüberprüfung wird nur vorgenommen, wenn die Compiler-Option **„Subrange testen"** während des Compilierens angewählt ist.

Division by zero.

Es wurde mit „/", „div" oder „mod" durch 0 dividiert.

Error: Can't open xxx.library

a) Beim Befehl OpenLib wurde die angegebene Bibliothek nicht gefunden.
b) Die „mathtrans.library", die für verschiedene Real-Funktionen benötigt wird, befindet sich nicht im Verzeichnis „libs:".

Out of memory.

Das freie RAM ist knapp, so daß *„New"* oder *„Alloc_mem"* nicht ausgeführt werden konnte.

Freemem failed.

a) Mit DISPOSE wurde versucht, ein dynamisches Objekt zu löschen, das nicht existiert bzw. bereits gelöscht wurde.

b) Es wurde versucht, mit Free_Mem Speicher freizugeben, der nicht mit dem Befehl Alloc_mem reserviert wurde (wenn Speicher mit der Exec-Funktion „AllocMem" angefordert wurde, kann er nur mit „FreeMem", NICHT mit dem Befehl „Free_Mem" freigegeben werden).

User Break.

Sie haben das Programm mit der Taste <F10> abgebrochen.

Error in Case.

In einem CASE-Statement ohne ELSE-Zweig trat eine nicht vorgesehene Alternative auf (also ein Wert, der nicht als Konstante angegeben ist).

Error in CloseLib

Die angegebene Library existiert nicht, wurde bereits geschlossen oder war nicht mit OpenLib (sondern z. B. mit der Exec-Funktion *„OpenLibrary"*) geöffnet worden.

Overflow error.

Es trat in einer arithmetischen Operation (z. B. Integer-Addition) ein Überlauf auf.

Um eine solche Fehlermeldung zu erhalten, müssen Sie vor dem Compilieren die Compiler-Option ***„Arithm. Überl."*** aktivieren.

Intuition error

Ein Window oder Screen konnte nicht geöffnet oder geschlossen werden.

Exec error

Bei einer Exec-Funktion lief etwas schief.

D) Die Kommandos des Editors

In diesem Teil des Anhangs erhalten Sie eine thematische Übersicht über alle verfügbaren Kommandos des Editors. Der Hinweis „Referenz" zu Beginn eines Abschnitts gibt Ihnen das jeweilige Kapitel an, in welchem Sie weitere Informationen zu dem vorgestellten Kommando erhalten können.

Alle Editorkommandos werden in einem einheitlichen Format dargestellt. Bei der Angabe der Parameter gilt es folgendes zu beachten :

Die Parameter werden von p^0, p^1 bis p^n durchnumeriert. Hinter jedem Parameter wird sein Typ angegeben.

Beispiel:

p^0=STRING

bedeutet, daß der erste Parameter ein String sein soll.

Es gibt folgende Parametertypen :

1) VALUE:

Eine ganze Zahl.

2) STRING:

Eine durch Anführungszeichen „ begrenzte Zeichenfolge.

3) LIST:

Eine durch „(" und „)" begrenzte Aufzählung von Elementen. Die Elemente dürfen dabei VALUES, STRINGS und natürlich auch LISTen sein. Im allgemeinen erwarten die Kommandos Edwards jedoch ihre Parameter in einer genau spezifizierten Reihenfolge. Informationen darüber sind dann aber in der Beschreibung des Befehls zu finden. Listen dienen ebenfalls der Rückgabe von Werten, insbesondere im Zusammenhang mit der ARexx-Schnittstelle.

Ein „OPT_" vor einem Parameter bedeutet, daß er optional ist und auch weggelassen werden darf.

Als Rückgabewert ist außer VALUE, STRING und LIST zusätzlich noch der sogenannte SMALL_STRING möglich. Hierbei handelt es sich um eine Zeichenfolge, die nicht durch Anführungsstriche begrenzt wird. Dies ist allerings nur im Zusammenhang mit der ARexx-Schnittstelle sinnvoll, da die Kommandos Edwards ihre String-Parameter immer als STRING erwarten.

D-1 Eingabe und Bearbeitung von Texten

Kommando : **BackSpace**
Abkürzung : **Back**
Parameter : -
Funktion :

Löscht das Zeichen links vom Cursor. Steht der Cursor am Anfang einer Zeile, so wird diese mit der vorigen Zeile zusammengefüg.

Kommando : **Delete**
Abkürzung : **Del**
Parameter : -
Funktion :

Löscht das Zeichen, auf dem sich der Cursor befindet. Steht der Cursor am Ende einer Zeile, so wird diese mit der folgenden Zeile zusammengefügt.

Kommando : **KillLine**
Abkürzung : **Kill**
Parameter : -
Funktion :

Entfernt die aktuelle Zeile aus dem Text. Sie wird im Undo-Buffer gespeichert.

Kommando : **UndoLine**
Abkürzung : **Undo**
Parameter : -
Funktion :

Befindet sich der Cursor in der zuletzt geänderten Zeile, so wird diese in ihren alten Zustand vor der Änderung versetzt.
Befindet sich der Cursor in einer anderen Zeile, so wird die zuletzt geänderte Zeile in ihrem Zustand vor der Änderung vor der aktuellen Zeile eingefügt. Dieses Verhalten dient vor allen dazu, eine durch „KillLine" gelöschte Zeile wieder richtig in den Text einzufügen.

Kommando : **IndentReturn**
Abkürzung : **IRet**
Parameter : -
Funktion :

Die aktuelle Zeile wird an der Cursorposition in zwei Zeilen aufgespalten. Der Teil der aktuellen Zeile, rechts vom Cursor , wird zur neuen aktuellen Zeile und mit Leerzeichen bzw. Tabs genau wie die davorliegende Zeile (die alte aktuelle) eingerückt.

Kommando : **Return**
Abkürzung : **Ret**
Parameter : -
Funktion :

Die aktuelle Zeile wird an der Cursorposition in zwei Zeilen aufgespalten. Der Teil der aktuellen Zeile, rechts vom Cursor, wird zur neuen aktuellen Zeile.

Kommando : **BigStepUp**
Abkürzung : **BSUp**
Parameter : -
Funktion :

Bewegt den Cursor um etwas weniger als die Anzahl der angezeigten Zeilen nach oben.

Kommando : **BigStepDown**
Abkürzung : **BSDown**
Parameter : -
Funktion :

Bewegt den Cursor um etwas weniger als die Anzahl der angezeigten Zeilen nach unten.

Kommando : **MoveUp**
Abkürzung : **Up**
Parameter : **p^0=VALUE**
MoFunktion :

Bewegt den Cursor um p^0 Zeilen nach oben.

Kommando : **MoveDown**
Abkürzung : **Down**
Parameter : **p⁰=VALUE**
Funktion :

Bewegt den Cursor um p⁰ Zeilen nach unten.

Kommando : **ScreenUp**
Abkürzung : **SUp**
Parameter : **p⁰=VALUE**
Funktion :

Verschiebt den Textausschnitt unter dem Cursor um p⁰ Zeilen nach unten.

Kommando : **ScreenDown**
Abkürzung : **SDown**
Parameter : **p⁰=VALUE**
Funktion :

Verschiebt den Textausschnitt unter dem Cursor um p⁰ Zeilen nach oben.

Kommando : **MoveLeft**
Abkürzung : **Left**
Parameter : **p⁰=VALUE**
Funktion :

Bewegt den Cursor um p⁰ Zeichen nach links. Geht der Cursor über den Anfang einer Zeile hinaus, so wird er an das Ende der davorliegenden Zeile gesetzt.

Kommando : **MoveRight**
Abkürzung : **Right**
Parameter : **p⁰=VALUE**
Funktion :

Bewegt den Cursor um p⁰ Zeichen nach rechts. Geht der Cursor über das Ende einer Zeile hinaus, so wird er an den Anfang der folgenden Zeile gesetzt.

Kommando : **GoStringPos**
Abkürzung : **GoSPos**
Parameter : **p^0=VALUE**
Return : **-**
Funktion :

Positioniert den Cursor auf das Zeichen an Position p^0 der Zeichenkette der aktuellen Zeile. Die Position wird von 0 an gezählt.

Kommando : **GoDispPos**
Abkürzung : **GoPos**
Parameter : **p^0=VALUE**
Return : **-**
Funktion :

Positioniert den Cursor in der aktuellen Zeile auf Position p^0. p^0 wird in diesem Fall von 1 an gezählt und korrespondiert mit der Anzeige in der Titelleiste des Fensters.

Kommando : **Top**
Abkürzung : **T**
Parameter : **-**
Funktion :

Setzt den Cursor auf den Anfang der ersten Zeile des Textes.

Kommando : **Bot**
Abkürzung : **B**
Parameter : **-**
Funktion :

Setzt den Cursor auf den Anfang der letzten Zeile des Textes.

D-2 Verwalten von Texten

Kommando : **NewText**
Abkürzung : -
Funktion :

Erzeugt einen neuen Text.

Kommando : **FreeText**
Abkürzung : **FTx**
Parameter : -
Funktion :

Gibt den aktuellen Text frei.

Kommando : **PredText**
Abkürzung : **PTx**
Parameter : -
Funktion :

Wählt den Text vor dem aktuellen Text an oder, falls dieser der erste Text in der Liste war, den letzten.

Kommando : **SuccText**
Abkürzung : **STx**
Parameter : -
Funktion :

Wählt den Text nach dem aktuellen Text an oder, falls dieserder letzte Text in der Liste war, den ersten.

Kommando : **SelTextN**
Abkürzung : **STN**
Parameter : **p⁰=VALUE**
Funktion :

Wählt Text Nummer p⁰ aus der Liste vorhandener Texte aus.

Kommando : **New**
Abkürzung : **New**
Parameter : **-**
Funktion :

Löscht alle Zeilen des aktuellen Textes.

Kommando : **Print**
Abkürzung : **Pr**
Parameter : **p^0=STRING**
Funktion :

Der aktuelle Text wird in die durch p^0 angegebene Datei ausgegeben. Durch Angabe von p0="ptr" wird auf den durch die Preferences eingestellten Drucker gedruckt.

D-3 Dateioperationen

Kommando : **Save**
Abkürzung : **SA**
Parameter : **p⁰=OPT_STRING**
Funktion :

Speichern des aktuellen Textes.
Wurde p⁰ angegeben, so wird der Text unter diesem Namen abgespeichert und als neuer Name des Textes übernommen. Wurde für p⁰ ein Leer-String angebenen, so wird der Name des Textes über einen Dateiauswahlfenster vom Benutzer abgefragt.
Wird p⁰ nicht mitangegeben, so wird der Text unter seinem bisherigen Namen abgespeichert.

Kommando : **Open**
Abkürzung : **Op**
Parameter : **p⁰=OPT_STRING**
Funktion :

Überladen des aktuellen Textes. Der aktuelle Text wird „vergessen", und der durch p⁰ angegebene Text wird geladen. Wird p⁰ nicht oder nur ein Leer-String für p⁰ angegeben, so wird der Name des Textes über einen Dateiauswahlfenster vom Benutzer abgefragt.

Kommando : **InsertFile**
Abkürzung : **InF**
Parameter : **p⁰=OPT_STRING**
Funktion :

Einfügen eines Textes in den aktuellen Text. Zu p⁰ gilt das bereits oben zu „Save" und „Open" Gesagte.

Kommando : **UpdateText**
Abkürzung : **UtX**
Parameter : **p⁰=Value**
Funktion :

Speichert den Text mit der ID p⁰, falls er geändert aber noch nicht gespeichert wurde.

Kommando : **UpdateTexts**
Abkürzung : **UtXs**
Parameter : -
Funktion :

Speichert jeden Texte, der geändert aber noch nicht abgespeichert wurde.

D-4 Blockoperationen

Kommando : **Mark**
Abkürzung : **Mk**
Parameter : -
Funktion :

Aktiviert den Markierungsmodus für normale Blöcke. Die Cursorposition bei Ausführung von „Mark" wird als Block-Anfang benutzt. Die aktuelle Cursorposition bei Ausführung einer Blockoperation gilt als Block-Ende.
Erneutes Ausführen von „Mark" bewirkt eine Desaktivierung des Markierungsmodus.

Kommando : **MarkVert**
Abkürzung : **MV**
Parameter : -
Funktion :

Aktiviert den Markierungsmodus für Spaltenblöcke. Erneutes Ausführen von „MarkVert" bewirkt eine Desaktivierung des Markierungsmodus.

Kommando : **Cut**
Abkürzung : **X**
Parameter : -
Funktion :

Schneidet dem markierten Block aus dem Text aus und kopiert ihn in das ClipBoard. Beendet den Blockmarkierungsmodus. Der Text vor dem ausgeschnittenen Bereich wird mit dem Text nach dem ausgeschnittenen Bereich zusammengefügt.

Kommando : **Copy**
Abkürzung : **C**
Parameter : **-**
Funktion :

Kopiert den markierten Block in das ClipBoard. Beendet den Blockmarkierungsmodus.

Kommando : **Paste**
Abkürzung : **V**
Parameter : **-**
Funktion :

Fügt den im ClipBoard gespeicherten Text an der momentanen Cursorposition ein. Anschließend wird der Cursor auf das letzte eingefügte Zeichen positioniert.

Kommando : **PasteVert**
Abkürzung : **PV**
Parameter : **-**
Funktion :

Fügt die im ClipBoard gespeicherte Textspalte korrekt an der momentanen Cursorposition ein. Anschliessend wird der Cursor auf das letzte eingefügte Zeichen positioniert.

Kommando : **SaveClip**
Abkürzung : **SC**
Parameter : **p⁰=OPT_STRING**
Funktion :

Speichert Text aus dem ClipBoard in einer Datei mit Namen p⁰. Wird p⁰ nicht oder nur ein Leer-String für p⁰ angegeben, so wird der Name der Datei über einen Dateiauswahlfenster vom Benutzer abgefragt.

Kommando : **OpenClip**
Abkürzung : **OC**
Parameter : **p⁰=OPT_STRING**
Funktion :

Lädt Text aus der durch p⁰ angebenen Datei und kopiert ihn in das ClipBoard. Wird p⁰ nicht oder nur ein Leer-String für p⁰ angegeben, so wird der Name der Datei über einen Dateiauswahlfenster vom Benutzer abgefragt.

Kommando : **BlockSL**
Abkürzung : **BSl**
Parameter : -
Funktion :

Rückt alle markierten Zeile eines Blocks um eine Tabulatorweite nach links, falls vor jeder Zeile mindestens ein Tab-Zeichen, bzw. entsprechend viele Leerzeichen sind.

Kommando : **BlockSR**
Abkürzung : **BSr**
Parameter : -
Funktion :

Rückt alle markierten Zeile eines Blocks um eine Tabulatorweite nach rechts.

Kommando : **BlockLower**
Abkürzung : **BLo**
Parameter : -
Funktion :

Wandelt alle markierten Zeichen eines Blocks in Kleinbuchstaben um.

Kommando : **BlockHigher**
Abkürzung : **BHi**
Parameter : -
Funktion :

Wandelt alle markierten Zeichen eines Blocks in Großbuchstaben um.

D-5 Suchen und Ersetzen

Kommando : **Find**
Abkürzung : **f**
Parameter : **p^0=STRING**
Funktion :

Sucht ausgehend von der aktuellen Cursorposition die durch p^0 angebene Zeichenkette im Text. Dabei wird solange in eine Richtung gesucht bis ein Vorkommen der Zeichenkette gefunden werden konnte. Ansonsten wird die Suchrichtung umgekehrt und abgebrochen.

Kommando : **Replace**
Abkürzung : **r**
Parameter : **p^0=STRING,** p^1=STRING
Funktion :

Sucht ausgehend von der aktuellen Cursorposition die durch p^0 angebene Zeichenkette im Text. Dabei wird solange in eine Richtung gesucht bis ein Vorkommen der Zeichenkette gefunden werden konnte. Ansonsten wird die Suchrichtung umgekehrt und abgebrochen.
Konnte p^0 gefunden werden, so wird sie durch die mit p^1 angegebene Zeichenkette ersetzt.

Kommando : **Again**
Abkürzung : **a**
Parameter : **-**
Funktion :

Wiederholt die letzte Such- bzw. Ersetzungsoperation.

Kommando : **ReplaceAll**
Abkürzung : **ra**
Parameter : **p^0=STRING, p^1=STRING**
Funktion :

Ersetzt alle Vorkommen von p^0 im Text durch die Zeichenkette p^1.

Kommando : **RemTab**
Abkürzung : **rt**
Parameter : **-**
Funktion :

Ersetzt alle Tabs im Text durch entsprechend viele Spaces.

Kommando : **GoMatchStruct**
Abkürzung : **GMS**
Parameter : **-**
Funktion :

Der Cursor muß sich vor Ausführung von „GoMatchStruct" auf einer beliebigen öffnenden oder schließenden Klammer befinden. „GoMatchStruct" bewegt den Cursor dann auf die korrespondieren Klammer.

Kommando : **GoParsedError**
Abkürzung : **GPE**
Parameter : **-**
Funktion :

Der Cursor muß sich auf der Zeile einer von einem Compiler erzeugten Fehlerdatei befinden. in welcher der Dateiname und die Zeilennummer eines Fehlers stehen. „GoParsedError" analysiert diese Zeile und prüft, ob die Datei, zu der die Fehlerbeschreibung gehört, sich schon im Speicher befindet. Ist dies nicht der Fall, so wird die entsprechende Datei nachgeladen. Daraufhin wird der Cursor auf die Zeile positioniert, in welcher sich der angebene Fehler befindet.

Kommando : **Jump**
Abkürzung : **J**
Parameter : **p^0=VALUE**
Funktion :

Positioniert den Cursor auf die durch p^0 angegebene Zeile.

D-6 Strukturieren von Texten

Kommando : **Fold**
Abkürzung : **Fo**
Parameter : -
Funktion :

Faltet einen markierten Block zusammen.

Kommando : **ShowFold**
Abkürzung : **SF**
Parameter : -
Funktion :

Befindet sich der Cursor auf einer geschlossenen Falte, so wird diese hierdurch geöffnet.

Kommando : **RemFold**
Abkürzung : **rf**
Parameter : -
Funktion :

Befindet sich der Cursor auf der ersten Zeile einer Falte, so wird die Falte durch „RemFold" geöffnet, und die Faltenmarkierungen werden entfernt.

Kommando : **SetFoldsOff**
Abkürzung : **SF0**
Parameter : -
Funktion :

Sorgt dafür, daß geschlossene, gefaltete Bereiche mit allen Zeilen genau wie geöffnete, gefaltete Bereiche dargestellt werden.

Kommando : **SetFoldsOn**
Abkürzung : **SF1**
Parameter : -
Funktion :

Macht „SetFoldsOff" rückgängig. Danach werden nur noch explizit mit „ShowFold" geöffnete Falten auch als geöffnet dargestellt.

Kommando : **GoPredFold**
Abkürzung : **GPF**
Parameter : -
Funktion :

Bewegt den Cursor auf den nächstliegenden Faltenanfang oder das nächstliegende Faltenende vor der aktuellen Zeile.

Kommando : **GoSuccFold**
Abkürzung : **GSF**
Parameter : -
Funktion :

Bewegt den Cursor auf den nächstliegenden Faltenanfang oder das nächstliegende Faltenende nach der aktuellen Zeile.

Kommando : **ToggleKey**
Abkürzung : **TK**
Parameter : -
Funktion :

Ändert den Auszeichnungsstatus der aktuellen Zeile. Ist diese ausgezeichnet, so wird die Auszeichnung gelöscht und umgekehrt.

Kommando : **GoPredKey**
Abkürzung : **GPK**
Parameter : -
Funktion :

Bewegt den Cursor auf die nächste vor der aktuellen Zeile liegende Zeile, die ausgezeichnet ist.

Kommando : **GoSuccKey**
Abkürzung : **GSK**
Parameter : -
Funktion :

Bewegt den Cursor auf die nächste der aktuellen Zeile folgende Zeile, die ausgezeichnet ist.

Kommando : **SetBookMark**
Abkürzung : **SBM**
Parameter : **p⁰=VALUE**
Funktion :

Belegt das Lesezeichen mit der Nummer p⁰ mit einem Verweis auf die aktuelle Zeile, so daß diese einfach wieder angesprungen werden kann.

Kommando : **GoBookMark**
Abkürzung : **GBM**
Parameter : **p⁰=VALUE**
Funktion :

Sucht, ob das Lesezeichen mit der Nummer p⁰ einen Verweis auf eine Zeile enthält und positioniert dann den Cursor auf diese.

D-7 Ändern von Optionen für den aktuellen Text

Kommando : **SetTab**
Abkürzung : **-**
Parameter : **p^0=VALUE**
Funktion :

Setzt die Tabulator-Weite auf p^0. p^0 muß im Bereich von 1 bis 20 liegen.

Kommando : **SetOverOn**
Abkürzung : **SO1**
Parameter : **-**
Funktion :

Aktiviert den „OverWrite"-Modus.

Kommando : **SetOverOff**
Abkürzung : **SO0**
Parameter : -
Funktion :

Desaktiviert den „OverWrite"-Modus.

Kommando : **SetWrap**
Abkürzung : **SWr**
Parameter : **p^0=VALUE**
Funktion :

Setzt den rechten Rand für den Umbruch im „Word-Wrap"-Modus auf p^0. p^0 muß größer als 9 sein.

Kommando : **SetWrapOn**
Abkürzung : **sw1**
Parameter : **-**
Funktion :

Aktiviert den „Word-Wrap"-Modus.

Kommando : **SetWrapOff**
Abkürzung : **sw0**
Parameter : -
Funktion :

Desaktiviert den „Word-Wrap"-Modus.

D-8 Word-Wrap

Kommando : **WrapPara**
Abkürzung : **WrP**
Parameter : -
Funktion :

Formatiert den aktuellen Absatz gemäß den Einstellungen für den „Word-Wrap" um.

Kommando : **WrapText**
Abkürzung : **WrT**
Parameter : -
Funktion :

Formatiert den aktuellen Text gemäß den Einstellungen für den „Word-Wrap" um.

D-9 Makros

Kommando : **RecStart**
Abkürzung : **rst**
Parameter : -
Funktion :

Leitet die Aufzeichnung eines Makros ein. Das zuletzt aufgezeichnete Makro geht dadurch verloren.

Kommando : **RecStop**
Abkürzung : **rsp**
Parameter : -
Funktion :

Beendet die Aufzeichnung eines Makros.

Kommando : **RecPlay**
Abkürzung : **RP**
Parameter : -
Funktion :

Spielt ein aufgezeichnetes oder eingeladenes Makro ab.

Kommando : **SaveRecScript**
Abkürzung : **SRS**
Parameter : **p^0=OPT_STRING**
Funktion :

Speichert ein aufgezeichnetes Makro in einer Datei mit Namen p^0 ab.
Wird p^0 nicht oder nur ein Leer-String für p^0 angegeben, so wird der Name der Datei über einen Dateiauswahlfenster vom Benutzer abgefragt.

Kommando : **PlayRecScript**
Abkürzung : **PRS**
Parameter : **p^0=OPT_STRING**
Funktion :

Lädt die durch p⁰ angegebene Makro-Datei ein, interpretiert diese und führt sie aus. Wird p⁰ nicht oder nur ein Leer-String für p⁰ angegeben, so wird der Name der Datei über einen Dateiauswahlfenster vom Benutzer abgefragt.

D-10 Fenster

Kommando : **NewWin**
Abkürzung : **NW**
Parameter : -
Funktion :

Eröffnet ein neues Fenster.

Kommando : **FreeWin**
Abkürzung : **FW**
Parameter : -
Funktion :

Schließt das aktuelle Fenster. Falls das aktuelle Fenster das letzte Fenster Edwards war, so wird Edward beendet.

Kommando : **PredWin**
Abkürzung : **PW**
Parameter : -
Funktion :

Wählt das Window vor dem aktuellen Window an oder, falls dieses das erste Window in der Liste war, das letzte.

Kommando : **SuccWin**
Abkürzung : **SW**
Parameter : -
Funktion :

Wählt das Window nach dem aktuellen Window an oder, falls dieses das letzte Window in der Liste war, das erste.

Kommando : **SplitHorWin**
Abkürzung : **SHW**
Parameter : -
Funktion :

Teilt das aktuelle Fenster horizontal in zwei gleichgroße Fenster auf.

Kommando : **SplitVerWin**
Abkürzung : **SVW**
Parameter : -
Funktion :

Teilt das aktuelle Fenster vertikal in zwei gleichgroße Fenster auf.

Kommando : **FullSizeWin**
Abkürzung : **FSW**
Parameter : -
Funktion :

Vergrößert das aktuelle Fenster auf maximale Ausmaße.

D-11 Verlassen des Editors

Kommando : **Exit**
Abkürzung : **XIt**
Parameter : -
Funktion :

Edward verlassen. Informationen zur Aufnahme der Arbeit in derselben Umgebung, wie vor dem Aufruf von „Exit" werden gespeichert.

Kommando : **Quit**
Abkürzung : **Q**
Parameter : -
Funktion :

Edward sofort verlassen.

D-12 Formulare

Kommando : **FindForm**
Abkürzung : **FFm**
Parameter : -
Funktion :

Öffnet das Such-Formular.

Kommando : **FindRepForm**
Abkürzung : **FRFm**
Parameter : -
Funktion :

Öffnet das Formular zum Suchen und Ersetzen.

Kommando : **GotoForm**
Abkürzung : **GFm**
Parameter : -
Funktion :

Öffnet das Formular, mit welchem eine spezielle Zeile angesprungen werden kann.

Kommando : **GlobalForm**
Abkürzung : **GFm**
Parameter : -
Funktion :

Öffnet das Formular für die globalen Einstellungen.

Kommando : **PageForm**
Abkürzung : **PFm**
Parameter : -
Funktion :

Öffnet das Formular, in welchem den Ausdruck von Texten betreffende Einstellungen vorgenommen werden können.

Kommando : **EditForm**
Abkürzung : **EdFm**
Parameter : -
Funktion :

Öffnet das Formular, mit dem den aktuellen Text betreffende Einstellungen vorgenommen werden können.

Kommando : **InfoForm**
Abkürzung : **InfFm**
Parameter : -
Funktion :

Öffnet das Formular mit Informationen über Edward.

D-13 Der Kommandozeilen-Interpreter

Kommando : **RepeatCmd**
Abkürzung : **RpCmd**
Parameter : -
Funktion :

Führt die zuletzt eingegebene Kommandozeile erneut aus.

Kommando : **EnterCmd**
Abkürzung : **ECmd**
Parameter : -
Funktion :

Wechselt in den Eingabemodus für die Kommandozeile. Ist dieser Eingabemodus aktiv, so bricht erneutes Ausführen von „EnterCmd" den Eingabemodus ab, ohne daß die eingebene Zeile ausgeführt wird.

Kommando : **ModifyOldCmd**
Abkürzung : **MCmd**
Parameter : -
Funktion :

Wechselt in den Eingabemodus für die Kommandozeile, wobei die zuletzt eingegebene Kommandozeile erneut zur Bearbeitung bereitgestellt wird. Ist dieser Eingabemodus aktiv, so bricht erneutes Ausführen von „ModifyOldCmd" den Eingabemodus ab, ohne daß die eingebene Zeile ausgeführt wird.

Kommando : **Do**
Abkürzung : **Do**
Parameter : **p^0=VALUE,** p^1=CMDLIST
Funktion :

Führt die durch p^1 angegebene Kommandofolge p^0-mal aus.

D-14 Ändern von globalen Einstellungen

Kommando : **Display**
Abkürzung : **Dsp**
Parameter : **p^0=VALUE**
Funktion :

Durch „Display" wird der Screen gewählt, auf welchem die Fenster von Edward erscheinen sollen. Dabei sind die Werte von p^0 folgendermaßen zugeordnet :

0 = Workbench-Screen

1 = 4 farbiger Screen, Hires

2 = 4 farbiger Screen, Hires-Interlaced

Übrigens - sollten Sie einmal einen anderen Screen als den Workbench-Screen eingeschaltet haben, bekommen Sie Edward nur mit dem Kommando DSP 0 wieder zurück auf den Workbench-Screen.

Kommando : **Font**
Abkürzung : **Fnt**
Parameter : **p^0=STRING, p^1=VALUE**
Funktion :

p^0 muß den Namen eines Zeichensatzes angeben (Bsp. „topaz.font"). p^1 gibt die gewünschte Größe des Zeichensatzes an. Bei der Auswahl der Zeichensätze gelten gewisse Einschränkungen. So sind z.B. keine proportionalen Zeichensätze möglich. Sollte Edward aus irgendeinem Grund einen gewünschten Zeichensatz für ungeeignet halten, so wird der zuvor eingestellte Zeichensatz weiterhin verwendet.

Kommando : **AutoMode**
Abkürzung : **Auto**
Parameter : **p^0=VALUE, p^1=OPT_VALUE**
Funktion :

p^0 wählt das Verfahren für den Auto-Save, p^1 gibt optional die Zeitspanne in Sekunden zwischen zwei Sicherungen an. p^1 muß einen Wert größer oder gleich 30 haben.

Die Werte von p^0 entsprechen folgenden Modi :

0 = Auto-Save OFF

1 = Auto-Save (automatisch speichern, ohne Bestätigung)

2 = Confirm Auto-Save (automatisch speichern, aber erst nach Bestätigung)

Kommando : **OpenDefs**
Abkürzung : **OD**
Parameter : **p^0=OPT_STRING**
Funktion :

Die Definition der Menüs, Tastaturbelegung, des HotKeys, etc. wird aus der durch den String p0 spezifizierten Datei geladen. Wird kein String angegeben, so öffnet sich ein Dateiauswahlfenster, mit dem eine Datei gewählt werden kann.

D-15 ARexx - Kommunikation mit Edward

Kommando : **ExecRexx**
Abkürzung : **RX**
Parameter : **p⁰=OPT_STRING**
Funktion :

Führt die durch p⁰ angebene Datei als Rexx-Programm aus. Wird p⁰ nicht oder nur ein Leer-String für p⁰ angegeben, so wird der Name der Datei über einen Dateiauswahlfenster vom Benutzer abgefragt.

Kommando : **ExecRexxString**
Abkürzung : **RXS**
Parameter : **p⁰=STRING**
Funktion :

Führt den übergebenen String p⁰ als kleines Rexx-Programm aus.

D-15.1 Abfrage von Statuswerten Edwards :

Kommando : **GetWord**
Abkürzung : **GWd**
Parameter : **-**
Return : **SMALL_STRING**
Funktion :

Liefert das Wort, auf welchem sich der Cursor befindet zurück.

Kommando : **GetLineString**
Abkürzung : **LineString**
Parameter : **-**
Return : **STRING**
Funktion :

Gibt den Inhalt der aktuellen Zeile zurück.

Kommando : **GetTextPos**
Abkürzung : **TextPos**
Parameter : -
Return : **LIST „(LineNum, DispPos, DispLineNum, EdStringPos, Fold, FoldStart, FoldEnd, Key, EndLF, EndEOS)“**
Funktion :

Liefert eine Liste, die die Position im Dokument und Status-Informationen enthält.

Aufbau der Liste :

- Zeilennummer; ergibt sich durch Abzählen aller Zeilen vom Textanfang an.
- Spaltenposition
- Zeilennummer; ergibt sich durch Abzählen nur der sichtbaren Zeilen (d.h.: die eingefalteten Zeilen werden nicht mitgezählt).
- Position im String der aktuellen Zeile, die mit der Spaltenposition korrespondiert. Enthält ein String TAB-Zeichen, so können sich die Spaltenposition und die String-Position unterscheiden, da ein Tab für die Spaltenposition normalerweise wie mehrere Leerzeichen gezählt wird.
- Fold. Flag; gibt an, ob die Zeile eingefaltet ist.
- FoldStart. Flag; gibt an, ob die Zeile der Beginn einer Falte ist.
- FoldEnd. Flag; gibt an, ob die Zeile das Ende einer Falte ist.
- Key. Flag; gibt an, ob die Zeile ausgezeichnet wurde.
- EndLF. Flag; gibt an, ob die Zeile mit einem LineFeed (ASCII 10) endet.
- EndEOS. Flag; gibt an, ob die Zeile mit einem EOS (ASCII 0) endet.

Kommando : **GetTextList**
Abkürzung : **TextList**
Parameter : -
Return : **LIST „([TextName ID]*)“**
Funktion :

Liefert eine Liste aller geladen Texte und ihrer IDs.

Kommando : **GetWindowList**
Abkürzung : **WindowList**
Parameter : -
Return : **LIST „([WindowID TextID Left Top Width Height]*)“**
Funktion :

Liefert eine Liste mit IDs aller geöffneten Windows, sowie dazugehörigen Informationen (ID des dargestellten Textes, Position und Ausmaße des Fensters).

Kommando : **GetBlockInfo**
Abkürzung : **BlockInfo**
Parameter : -
Return : **LIST „(Block, MarkVert, LineNum, DispPos, StringPos)“**
Funktion :

Gibt Informationen über einen möglicherweise markierten Block zurück. Ein markierter Block wird immer durch die zu Beginn markierte Zeile und die aktuelle Zeile definiert.

Die Liste enthält folgende Werte:

- Block; Flag. Gibt an, ob ein Block markiert ist. Ist dieses Flag 0, so gelten alle folgenden Informationen nicht.
- MarkVert; Flag. Gibt an, daß der Benutzer einen Spaltenblock markieren will.
- LineNum. Zeilennummer der Zeile, an der die Markierung beginnt (diese kann über-, aber auch unterhalb der aktuellen Zeile liegen).
- DispPos. Spaltenposition des Markierungsbeginns.
- StringPos. Zu DispPos korrespondierende Position im String der Zeile, an der die Markierung beginnt.

Kommando : **GetTextInfo**
Abkürzung : **TextInfo**
Parameter : -
Return : **LIST „(TotalLines, DispLines, FoldedLines, UseFold, DoWrap, Over, LockCnt, Changes.)“**
Funktion :

Liefert allgemeine Informationen über den aktuellen Text.

Aufbau der Liste :

- TotalLines; Anzahl aller Zeilen des Textes
- DispLines; Anzahl der momentan darstellbaren Zeilen.
- FoldedLines; Anzahl der eingefalteten Zeilen.
- UseFold; Flag. Gibt an, ob Zeilen innerhalb von Falten dargestellt werden sollen (0) oder nicht(-1).
- DoWrap; Flag. Gibt an, ob der WordWrap-Modus eingeschaltet ist.
- Over; Flag. Gibt an, ob der OverWrite-Modus eingeschaltet ist.

- LockCnt; Zähler, wie oft der Text gelocked wurde. Ist er ungleich 0, so kann der Text nicht modifiziert werden. Der LockCnt wird hochgezählt, wenn der Text gedruckt werden soll, oder TextReads auf den Text ausgeführt werden.

Kommando : **GetScreenInfo**
Abkürzung : **ScreenInfo**
Parameter : -
Return : **LIST „(Width Height Depth ScreenAdr)"**
Funktion :

„ScreenInfo" gibt folgende Werte in einer Liste zurück :
Breite, Höhe, Tiefe und Adresse des Screens, auf welchem Edward seine Fenster darstellt.

D-15.2 Laden von Texten aus Edward heraus

Mit den folgenden Befehlen ist ein Zugriff auf im Speicher befindliche Texte Edwards auf einfache und schnelle Weise möglich.

Zu beachten ist hier vor allem, daß ein von einer externen Applikation mit InitTextRead geöffneter Text bis zum Abschluss der TextRead- Operationen durch EndTextRead nicht verändert werden kann; der Text ist gesperrt („gelocked").

Solange nicht alle InitTextReads durch entsprechende EndTextReads abgeschlossen wurden, kann zudem Edward nach einem Beenden nicht aus dem Speicher entfernt werden.

Es ist selbstverständlich möglich, mehrere InitTextReads auf denselben Text auszuführen.

Kommando : **InitTextRead**
Abkürzung : **ITR**
Parameter : **p^0=STRING**
Return : **Handle=VALUE**
Funktion :

Versucht Text mit Namen p0 zu finden. Ist dieser vorhanden, so wird er gelocked und ein Handle für weitere Operationen erzeugt. Die erste Zeile des Textes wird zur aktuellen Zeile für kommende TextRead-Operationen. Das Handle wird zurückgegeben.

Kommando : **TextRead**
Abkürzung : **TR**
Parameter : **Handle=VALUE, Buffer=VALUE, BuffSize=VALUE**
Return : **RealRead=VALUE**
Funktion :

Kopiert von der für diesen TextRead aktuellen Zeile den Inhalt dieser und der folgenden Zeilen in den Buffer. Dabei werden Zeilen immer komplett kopiert. Sollte eine Zeile nicht mehr ganz in den Buffer passen (daher muß BuffSize angegeben werden), so wird die Kopieraktion beendet.
Daraufhin wird die Anzahl der kopierten Bytes zurückgegeben (RealRead). Das Lesen ist beendet, wenn RealRead gleich 0 ist.

Kommando : **EndTextRead**
Abkürzung : **ETR**
Parameter : **Handle=VALUE**
Return : -
Funktion :

Der durch ein „InitTextRead" gelockte Text wird wieder unlocked. Das Handle wird freigegeben.

D-16 UnDo/ReDo - Zurücknehmen von Textänderungen

Kommando : **UnDo**
Abkürzung : **UnDo**
Parameter : -
Funktion :

Nimmt die letzte Änderung am Text zurück.

Kommando : **ReDo**
Abkürzung : **ReDo**
Parameter : -
Funktion :

Nimmt die Rücknahme der Rückname der letzten Textänderung zurück.
(Gut, das ist nicht gerade leichverständlich, aber Sie können ja noch einmal in das Referenzkapitel blättern. Noch besser ist, Sie probieren es einfach aus.)

D-17 Verschiedenes

Kommando : **ExecCli**
Abkürzung : **Exe**
Parameter : **p⁰=STRING**
Funktion :

Führt das Programm mit Namen p⁰ aus.

Kommando : **InsertChars**
Abkürzung : **IC**
Parameter : **p⁰=STRING**
Funktion :

Fügt die durch p⁰ spezifizierten Zeichen an der aktuellen Cursorposition ein. Dieses Kommando wurde für Makros und die ARexx-Kommunikation eingeführt.

Kommando : **SetMode1, SetMode2, SetMode3**
Abkürzung : **SM1, SM2, SM3**
Parameter : **-**
Funktion :

Sorgt dafür, daß die nächste Tastatureingabe gemäß Modus 1 interpretiert wird.

Kommando : **Iconify**
Abkürzung : **Ify**
Parameter : -
Funktion :

Schließt alle Fenster Edwards. Ein neues Fenster, das nur aus einer Titelzeile besteht erscheint anschließend. Wird dieses Fenster aktiviert und die rechte Maustaste gedrückt, so werden alle Fenster Edwards wieder an ihren ursprünglichen Positionen geöffnet. Edward kann ebenfalls durch den HotKey oder das Senden einer Nachricht über ARexx „geweckt" werden.

E) Editor-Fehlermeldungen.

Beim Arbiten mit Edward werden Sie bei wichtigen Problemen, die sich ergeben können, augenblicklich informiert und erhalten in den meisten Fällen die Möglichkeit, auf diese Informationen zu reagieren. In der Regel handelt es sich hier um Sicherheitsabfragen, um zu verhindern, daß Sie z.B. aus Versehen einen noch nicht abgespeicherten Text löschen.

E-1 Informationen

Meldung : **„Markierung gesetzt"**

Ursache : Sie haben die aktuelle Zeile mit einem Lesezeichen belegt, um sie demnächst leicht wieder anspringen zu können.

Referenz : Kapitel 3.7.4 Lesezeichen

E-2 Allgemeine Fehlermeldungen

Meldung : **„Text ist geschützt"**

Ursache : Sie wollten den Text verändern, dieser ist aber gesperrt. Ein Text wird gesperrt, wenn er gedruckt werden soll, oder der Befehl „InitTextRead" auf ihn ausgeführt wurde.

Abhilfe : Warten, bis der Text gedruckt wurde.
Vielleicht wird auch gerade ein ARexx-Makro auf diesen Text ausgeführt, dann heißt es ebenfalls abwarten.

Referenz : Kapitel IV. 5) Drucken eines Textes

Kapitel 3.17 ARexx - Kommunikation mit Edward

Meldung : **„Kein Pictogramm vorhanden"**

Ursache : Wenn Sie im Formular für die globalen Einstellungen „Configuration Icon" gewählt haben, versucht Edward zu jedem Text, den Sie erzeugen, eine „.info"-Datei zu schreiben. Diese Datei enthält unter anderem Informationen, damit die Workbench ein Pictogramm zu Ihrem Text darstellen kann. Existierte bisher keine „.info"-Datei, so erzeugt Edward eine Standard-Datei. Dazu benötigt er im Verzeichnis „Edward:" ie Datei „EdwardDefault.info". Diese haben Sie anscheinend gelöscht.

Abhilfe : Kopieren Sie von der mitgelieferten Diskette die Datei „EdwardDefault.info" wieder in Ihr „Edward:"-Verzeichnis, sollte Ihnen das zu kompliziert erscheinen müssen Sie Edward komplett neu installieren (Achtung: Dabei können möglicherweise verschiedene von Ihnen vorgenommene Voreinstellungen für Edward verloren gehen).

Referenz : Kapitel 3.15 „Globale Einstellungen"

Meldung : **„Abbruch! Konnte Backup nicht erzeugen!"**

Ursache : Sie wollten einen Text speichern. Beim Anlegen der Backup-Datei Ihres Textes trat ein Fehler auf.
Möglicherweise haben Sie einen ungültigenPfad für die Backup-Datei angegeben.
Vielleicht ist auch nur das Laufwerk oder die Platte voll, auf welcher Sie die Backup-Datei anlegen lassen wollten.

Abhilfe : Überprüfen Sie den Pfad-Namen, den Sie für die Backup-Datei im „Global Prefs"-Formular angegeben haben.
Versichern Sie sich, daß auf dem durch den Pfad angegebenen Medium genügend Speicherplatz vorhanden ist.

Referenz : Kapitel 3.15 „Globale Einstellungen"

Meldung : **„Markierung nicht gesetzt!"**

Ursache : Sie haben versucht ein Lesezeichen anzuspringen, das noch nicht definiert wurde.

Abhilfe : Vielleicht haben Sie nur die falsche Taste gedrückt, dann sollten Sie es einfach nochmal versuchen.

Referenz : Kapitel 3.7 „Strukturieren von Texten - Lesezeichen" Abschnitt 4)

Meldung : **„Kein Block definiert!"**

Ursache : Sie haben versucht eine Block-Operation auszuführen (Cut, Copy, Paste, etc.) , zuvor jedoch keinen Block definiert.

Abhilfe : Einfach den gewünschten Text-Bereich markieren und nochmal versuchen.

Referenz : Kapitel 3.5. „Blockoperationen"

Meldung : **„Abbruch! Funktion würde Einfaltung zerstören!"**

Ursache : Im allgemeinen wird diese Meldung ausgegeben, wenn Sie einen Block ausschneider wollten, der nur teilweise eine Text-Falte überdeckt. Bei einer solchen Operation wür-

de nur der Anfang bzw. das Ende der Textfalte ausgeschnitten, was die Falte zerstören würde. Deshalb unterbindet Edward solche Aktionen.

Abhilfe : Sie können eine Textfalte nur komplett ausschneiden. Sollten Sie tatsächlich nur einen Teil der Textfalte ausschneiden wollen, so müssen Sie die Faltung erst rückgängig machen, bevor Sie diesen Teil ausschneiden können.

Referenz : Kapitel 3.5 „Blockoperationen"

Meldung : **„Zeichenkette zu lang!"**

Ursache : Aus internen Gründen kann Edward maximal 512 Zeichen in einer Zeile verwalten. Beim Einfügen von Zeichen durch eine „Paste"-Operation könnte diese Grenze möglicherweise überschritten werden, was fatale Fehler hervorrufen würde. Daher stoppt Edward hier das Einfügen und gibt obige Meldung aus.

Abhilfe : Die „Paste"-Operation in einer leeren Zeile ausführen.

Referenz : Kapitel 3.5 „Blockoperationen"

Meldung : **„Suchmuster nicht gefunden!"**

Ursache : Beim Suchen oder Ersetzen konnte das von Ihnen angegebene Suchmuster nirgendwo im Text entdeckt werden.

Abhilfe : Suchvorgang wiederholen. Edward durchsucht den Text immer von der aktuellen Position aus entweder zum Anfang oder zum Ende hin nach dem Suchmuster. Konnte dieses nicht gefunden werden, so kehrt er die Suchrichtung um. Daher kann unter Umständen nach dem Fehlschlagen einer Suchaktion durch eine Wiederholung des Suchvorgangs doch noch etwas gefunden werden. Schlägt dieser Suchvorgang ebenfalls fehl. Dann ist beim besten Willen nichts zu finden.
Vielleicht verbirgt sich eine Zeile, deren Inhalt dem Suchmuster entsprechen würde, in einer Textfalte, dann konnte Edward hier ebenfalls nichts finden. Wählen Sie „Show All Folds" aus dem „Folds"-Menü und wiederholen Sie den Suchvorgang.

Referenz : Kapitel 4.6. „Suchen und Ersetzen"

Meldung : **„Cursor steht auf keinem Strukturelement"**

Ursache : Beim Aufruf von „Match Bracket" aus dem „Search"-Menü stand der Cursor auf keiner Klammer.

Abhilfe : Positionieren Sie den Cursor auf eine Klammer. Wählen Sie dann erneut „Match Bracket".

Referenz : Kapitel 3.6. „Suchen und Ersetzen"

Meldung : **„Strukturen stimmen nicht überein!"**

Ursache : Nach dem Aufruf von „Match Bracket" aus dem „Search"-Menü konnte Edward keine korrespondierende öffnende bzw. schließende Klammer finden.

Abhilfe : Überprüfen Sie Ihren Text genau. Sie haben sicher irgendwo eine Klammer vergessen.

Referenz : Kapitel 3.6 „Suchen und Ersetzen"

Meldung : **„Alles ist rückgängig gemacht worden!"**

Ursache : Sie haben „Undo" aus dem „Extras"-Menü so oft ausgewählt, daß es nichts mehr rückgängig zu machen gibt.

Abhilfe : Wählen Sie doch mal eine andere Funktion aus und bearbeiten Sie Ihren Text wieder.

Na ja, ´mal ganz ernsthaft. Die obige Meldung ist eigentlich eher als Hinweis denn als Fehler zu werten. Nehmen Sie sie einfach zur Kenntnis.

Referenz : Kapitel 3.18 „Zurücknehmen von Text-Änderungen"

Meldung : **„Alles ist wiederhergestellt worden!"**

Ursache : Sie haben alle Rücknahmen von Text-Änderungen wieder zurückgenommen, das heißt, Ihr Text sieht wieder genauso aus, wie vor dem ersten Aufruf von „Undo".

Abhilfe : Siehe oben.

Referenz : Kapitel 3.18 „Zurücknehmen von Text-Änderungen"

Meldung : **„Menü ist zu groß für den Screen! Nutze Default Einstellungen!"**

Ursache : Ihr Systemfont ist derart hoch, daß meist die Höhe des Menüs (in seltenen Fällen wird das Menü so breit, daß es in der horizontalen nicht mehr paßt) größer wird, als auf dem Screen angezeigt werden kann.

Abhilfe : Sie sollten keine Übertrieben großen Fonts verwenden, um noch etwas auf Ihrem Monitor erkennen zu können. Kaufen Sie sich besser einen größeren Monitor!

E-3 Fehlermeldungen mit Abfragen

Meldung : **„[1][Vorsicht: Kein Speicher mehr!|Texte entfernen|oder Programm verlassen.][OK]"**

Ursache : Edward hat leider nicht genügend Speicher bekommen.

Abhilfe : Sicherheitshalber sollten Sie alle Texte erst einmal abspeichern. Wenn Sie dann einen oder mehrere der momentan nicht mehr benötigten Texte löschen, dann haben Sie für 's erste genügend Speicher zum Weiterarbeiten. Sollte diese Meldung öfter auftreten, dann ist der Kauf einer Speichererweiterung mehr als nur ratsam.

E-4 Meldungen mit Abfragen

Meldung : **„Geänderten Text verwerfen ?**

[Ja] [Nein]"

Ursache : Sie wollten eine Operation vornehmen, deren Ausführung dazu führen würde, daß der momentan aktuelle Text verloren geht (z.B.: „Open" aus dem „Projekt"-Menü oder „Neu" bzw. „Frei" aus dem „Text"-Menü).
Falls dieser Text von Ihnen verändert, aber nicht abgespeichert wurde, so erfolgt diese Meldung. Wenn Sie „Nein" wählen, so wird die Operation abgebrochen und Sie können den Text danach abspeichern, bei „Ja" geht dieser Text unwiederbringlich verloren.

Abhilfe : Vor einer der oben erwähnten Operationen den Text abspeichern. Dies ist allerdings nicht zwingend notwendig. Die obige Meldung dient ja gerade dazu, Schlimmeres zu verhindern, wenn Sie das Abspeichern einmal vergessen sollten.

Referenz : Kapitel 3.3 „Text", Kapitel 3.4 „Dateioperationen"

Meldung : **„Geänderten Text verwerfen ?**

[Ja] [Nein]"

Ursache : Sie wollten Edward beenden, haben aber noch nicht alle geänderten Texte abgespeichert.

Abhilfe : Wählen Sie „Ja", dann wird Edward beendet und alle von Ihnen bearbeiteten Texte werden nicht abgespeichert und gehen verloren. Da dies normalerweise nicht gewünscht ist sollten Sie „Nein" wählen und dann die entsprechenden Texte abspeichern.

Referenz : Kapitel 3.4 „Dateioperationen"

Meldung : **„Altes Pictogramm ersetzen ?**

[Ja] [Nein]"

Ursache : Sie haben eine Datei bearbeitet, zu der ein Icon existiert, das nicht von Edward erzeugt wurde. Dies ist z.B. möglich, wenn Sie eine Script-Datei eines anderen Programmes bearbeiten. Da viele Programme in der „.info"-Datei die das Icon enthält wichtige Zusatzinformationen speichern (diese können Sie sich durch einfaches Anklicken des Icons und Wahl des Menüpunktes „Info" aus einem Menü der Workbench anzeigen lassen) sollten Sie bei dieser Meldung vorsichtshalber „Nein" wählen. Nur wenn Sie ganz sicher sind, daß die „.info"-Datei keine wichtigen Informationen enthält wählen Sie „Ja", dann überschreibt Edward diese Datei mit seiner eigenen „.info"-Datei, die Informationen über das gerade bearbeitete Dokument enthält.

Abhilfe : Vergewissern Sie sich vor der Bearbeitung einer Datei, ob eine „.info"-Datei zu Ihr existiert und ob diese von Edward oder einem fremden Programm erzeugt wurde. Ist letzteres der Fall, so sollten Sie mit dem „Info"-Menü der Workbench erkunden, ob diese Datei irgendwelche wichtigen Zusatzinformationen enthält.

Referenz : Kapitel 3.4 „Dateioperationen",

Kapitel XVI. „Globale Einstellungen" Abschnitt 1) „Das 'Global Prefs'-Formular"

Meldung : **„Datei speichern : 'Name'**

[Ja | Nein]"

Ursache : Sie haben im „Global Prefs"-Formular die Option „Confirm Autosave" eingestellt. Dies bedeutet, daß Edward nach dem von Ihnen eingestellten Zeitintervall alle Texte überprüft und dann versucht, die geänderten Texte abzuspeichern. Vorher holt er sich jedoch von Ihnen über obige Meldung die Erlaubnis.

Abhilfe : Wenn Sie diese Abfrage stört, dann wählen Sie statt „Confirm Auto-Save" die Option „Auto-Save", dann werden geänderte Texte automatisch ohne Rückfrage gespeichert.

Referenz : Kapitel 3.15 „Globale Einstellungen" Abschnitt 1) „Das 'Global Prefs'-Formular"

Meldung : **„Datei existiert bereits**

Überschreiben?

[Ja] [Nein]"

Ursache : Vermutlich haben Sie nach Wahl des Menüpunktes „Speichern als..." aus dem „Projekt"-Menü im Datei-Auswahlfenster den Namen einer Datei gewählt, die schon existiert. Diese würde von momentan aktuellen Text überschrieben, wenn Sie „Ja" wählen und

ginge auf ewig verloren. Um dies zu verhindern erfolgt die Rückfrage Edwards. Wenn Sie sicher sind, daß sie die schon existieren Datei überschreiben wollen, dann wählen Sie „Ja". Normalerweise ist es wohl sinnvoller „Nein" zu wählen und sich die entsprechende Datei erst einmal genauer anzuschauen.

Referenz : Kapitel 3.4 „Dateioperationen"

E-5 Fehler bei Betriebssystemoperationen

Meldung : **„Kein Speicher mehr!"**

Ursache : Sie leiden unter Speicherverknappung. Edward kann in diesem Fall die von Ihnen gewünschte Operation nicht korrekt ausführen.

Abhilfe : Kurzfristig ist es sinnvoll für mehr Speicher zu sorgen, indem sie nicht mehr benötigte Programme die momentan ebenfalls laufen beenden. Weiterhin können Sie Texte, die Sie nicht weiter bearbeiten möchten, mit „Frei" aus dem „Text"-Menü wieder freigeben und hierdurch ebenfalls etwas Speicher gewinnen.
Langfristig empfiehlt sich wohl die Anschaffung einer RAM-Erweiterung.

Meldung : **„Kann Datei nicht öffnen"**

Ursache : Edward versuchte eine von Ihnen angegebene Datei zum Laden oder Speichern zu öffnen. Dieser Versuch schlug fehl. Eine mögliche Ursache währe, daß diese Datei momentan von einem anderen Programm bearbeitet wird. Dies ist zum Beispiel möglich, wenn gerade ein Compiler diese Datei übersetzt.

Abhilfe : In oben erwähntem Fall empfiehlt es sich, zu warten, bis der Compiler seine Arbeit beendet hat und dann erneut zu versuchen, diese abzuspeichern. Ansonsten sollten Sie versuchen die Datei unter einem anderen Namen, möglicherweise auch auf einem anderen Laufwerk abzuspeichern.

Meldung : **„Kann nicht speichern"**

Ursache : Beim Schreiben einer Datei schlug irgendetwas fehl.

Abhilfe : Überprüfen Sie, daß das Laufwerk auf welchem Sie die Datei speichern wollten genügen Speicherplatz bietet und kein Schreibschutz aktiviert ist. Sollte dies nicht der Fall sein, so wechseln Sie das Laufwerk, oder deaktivieren den Schreibschutz.

Meldung : **„Kann Info-Datei nicht speichern"**

Ursache : Das Betriebssystem konnte die „.info"-Datei zu Ihrem Text nicht erzeugen.

Abhilfe : s. oben.

Meldung : **„Fehler beim Lesen"**

Ursache : Während des Lesens einer Datei trat ein Fehler auf. Möglicherweise sind Daten auf dem entsprechenden Laufwerk defekt.

Abhilfe : Versuchen Sie mit einem Programm wie DiskDoctor zu retten, was noch zu retten ist. Viel Glück!
Wenn Sie wie empfohlen fleißig Backups Ihrer Arbeiten gemacht haben, dann dürften Sie in dieser Situation nicht viel verloren haben.

Meldung : **„Konnte Datei nicht speichern"**

Ursache : Beim speichern einer Datei schlug irgendetwas fehl.

Abhilfe : s. „Kann nicht speichern"

Meldung : **„Konnte Formular nicht öffnen"**

Ursache : Edward versuchte ein von Ihnen gewünschtes Formular zu öffnen. Dies schien dem Betriebssystem nicht möglich zu sein. Normalerweise ist ein Mangel an freiem Speicher (insbesondere CHIP-Memory) an diesem Verhalten schuld.

Abhilfe : Für freien Speicher sorgen (s. „Keinen Speicher mehr").

E-6 Eingabefehler

Meldung : **„Wrap-Position darf nicht kleiner als 10 sein!"**

Ursache : Sie haben wahrscheinlich in der Kommandozeile eine Wrap-Position kleiner als 10 Zeichen setzen wollen. Da dies nicht sehr sinnvoll ist hat Edward dies unterbunden.

Abhilfe : Eine Position größer oder gleich zehn Zeichen angeben.

Referenz : Kapitel 3.9 „Word-Wrap"

Meldung : **„Tabulatorwert nur zwischen 1 bis 20"**

Ursache : Sie haben versucht einen Wert für die Größe des Tabulatorabstandes anzugeben, der als nicht sinnvoll erachtet wird.

Abhilfe : Sinnvollen Wert zwischen Eins und Zwanzig angeben.

Referenz : Kapitel III. „Eingabe und Bearbeitung von Texten" Abschnitt 4) RETURN, ENTER und TAB
Kapitel 3.15 „Globale Einstellungen" Abschnitt 1) „Das 'Global Prefs'-Formular"

Meldung : **„Auflösungen nur im Bereich von 0 bis 2"**

Ursache : Sie versuchten einen ungültigen Wert für den Display-Typ anzugeben.

Abhilfe : Wert zwischen Null und Zwei angeben.

Referenz : Kapitel 3.15 „Globale Einstellungen" Abschnitt 1) „Das 'Global Prefs'-Formular"

Meldung : **„Zeiten kleiner 30 Sek. machen keinen Sinn!"**

Ursache : Die künstliche Intelligenz Edwards hält Intervalle unter dreissig Sekunden für die „Auto-Save"-Funktion für nicht besonders sinnvoll. Wahrscheinlich haben Sie einen Wert unter dreissig Sekunden nur aus Versehen angegeben.

Abhilfe : Werte größer dreissig angeben.

Referenz : Kapitel 3.15 „Globale Einstellungen" Abschnitt 1) „Das 'Global Prefs'-Formular"

Meldung : **„Automode-Bereich nur von 0 bis 2!"**

Ursache : Sie haben einen ungültigen Wert für den „Auto-Save"-Modus angegeben.

Abhilfe : Wert zwischen Null und Zwei angeben.

Referenz : Kapitel 3.15 „Globale Einstellungen" Abschnitt 1) „Das 'Global Prefs'-Formular"

E-7 Fehler in den Definitionsdateien

Als Referenz zu diesem Abschnitt empfiehlt es sich, das Kapitel XX. „Benutzerdefinierte Menüs, Tastaturbelegung und HotKey" zu lesen.

Meldung : **„Präprocessor-Befehl zu komplex"**

Ursache : In einer Definitons-Datei, welche die Menüs und die Tastaturbelegung Edwards enthält trat eine übermäßig komplexe Verschachtelung von „#if"-Statements auf. Da Edward hier nur eine Tiefe von maximal 16 beherrscht, hat er hier das Handtuch geworfen.

Abhilfe : Formulieren Sie ihre Definitons-Dateien etwas weniger komplex. Es erscheint schon fast unglaublich, daß Sie selbst diese Schachtelungstiefe verstehen können.

Meldung : **„#endif ohne #if"**

Ursache : Sie haben ein #endif in ihrer Datei, zu dem kein #if gehört.

Abhilfe : Überprüfen Sie die Datei noch einmal genau.

Meldung : **„Präprocessor-Befehl erwartet"**

Ursache : Sie haben in der Datei ein „#"-Zeichen an einer Stelle verwendet, an welcher der Präprozessor dieser bearbeitet. Dieser erwartet nach dem „#"-Zeichen ein Statement („if", „endif" oder „else").

Abhilfe : Überprüfen Sie die Datei. Entfernen Sie das Zeichen, oder ergänzen Sie das entsprechende Statement.

Meldung : **„TITLE erwartet"**

Ursache : Sie haben in der Datei versucht ein Menü zu definieren, daß jedoch keinen einzigen Titel enthält.

Abhilfe : Die entsprechende Stelle der Datei untersuchen. Vielleicht haben Sie den „TITLE" Befehl ja einfach nur vergessen. Sollten Sie zu den Puristen gehören, die vollständig auf Menüs verzichten können und alles mit der Tastatur machen, dann entfernen Sie auch noch den „MENU"-Befehl aus der Datei.

Meldung : **„Zu viele Menüeinträge"**

Ursache : Sie haben versucht, ein Menü mit mehr als zwanzig Einträgen zu erzeugen. Dies ist leider nicht möglich.

Abhilfe : Die Einträge auf mehrere Menüs verteilen.

Meldung : **„Zu viele Menütitel“**

Ursache : Sie haben versucht, mehr als 16 Menüs zu erzeugen. Dies ist leider nicht möglich.

Abhilfe : Beschränken Sie sich etwas in Ihren Menüs.

E-8 Fehler beim Analysieren von Kommandos

Fehlermeldungen in Definitions-Dateien oder in Kommandozeilen folgt immer eine Angaben der Position, an welcher der entsprechende Fehler auftrat. Die Fehlermeldungen haben daher folgenden Form:

1. **Fehler**

   In Zeile : Zeile x Column: Spalte

oder

2. **Fehler**

   In Spalte: Spalte

Meldung : **„IDENTIFIER erwartet“**

Ursache : Es wurde ein IDENTIFIER erwartet, aber nicht gefunden.
Ein IDENTIFIER beginnt mit einem Buchstaben, gefolgt von einer beliebigen Anzahl alphanumerischer Zeichen (also Buchstaben und Ziffern) gemischt mit dem Unterstrich „_“.

Meldung : **„STRING erwartet“**

Ursache : Es wurde ein STRING erwartet. Ein String beginnt mit einem Hochkomma („) gefolgt von einer beliebigen Anzahl Zeichen und wird mit einem Hochkomma abgeschlossen.

Meldung : **„NUMBER erwartet“**

Ursache : An dieser Stelle wurde eine Zahl erwartet.

Meldung : **„(erwartet“**

Ursache : Es wurde eine öffnende, runde Klammer erwartet.

Meldung : **„) erwartet“**

Ursache : Eine schliessende, runde Klammer wurde erwartet.

Meldung : **„; erwartet“**

Ursache : Es wurde eine Semikolon erwartet.

Meldung : **„Kein Befehl“**

Ursache : Es wurde ein Kommando aus Edwards Befehlsvorrat erwartet.

Abhilfe : Überprüfen Sie die Stelle. Wahrscheinlich haben Sie den Befehl nur falsch ausgeschrieben.

Meldung : **„LIST oder IDENTIFIER erwartet“**

Ursache : Es wurde eine LIST oder ein IDENTIFIER erwartet, aber nicht gefunden.
Eine LIST wird von runden Klammern umschlossen. Zwischen diesen können sich IDENTIFIER, STRINGs, NUMBERs, COMMANDs oder LISTs auch gemischt befinden.

Meldung : **„LOBJECT erwartet“**

Ursache : Es wurde ein LOBJECT erwartet. Ein LOBJECT kann ein STRING, eine NUMBER, ein COMMAND oder eine LIST sein.

E-9 Fehlermeldungen beim Auswerten von Kommandos

Meldung : **„LVAL erwartet“**

Ursache : Ein LVAL, das bedeutet eine Zahl wurde von einem Kommando als Argument erwartet.

Meldung : **„Zu viele Argumente“**

Ursache : Ein Kommando wurde mit zuvielen Argumenten versorgt.

Meldung : **„Falsches Argument“**

Ursache : Ein Kommando wurde mit falschen Argumenten versorgt.

F) Kurzübersicht

1.) Wort-Symbole (reserviert Pascal-Schlüsselwörter):

AND	ARRAY	BEGIN	CASE
CONST	DIV	DO	DOWNTO
ELSE	END	FILE	FOR
FUNCTION	GOTO	IF	IN
LABEL	MOD	NIL	NOT
OF	OR	PROCEDURE	PROGRAM
PACKED	RECORD	REPEAT	SE
SHL	SHR	THEN	TO
TYPE	UNTIL	VAR	WHILE
WITH	XOR		

Diese Liste entspricht (außer SHL, SHR und XOR) dem Jensen-Wirth-Standard.

2.) Direktiven:

ABSOLUTE	EXPORT	EXTERNAL	FORWARD
FROM	IMPLEMENTATION	IMPORT	INTERFACE
LIBRARY	MODULE	OTHERWISE	STATIC STRING
UNIT	USES		

Der Jensen-Wirth- (ISO 7185-) Standard kennt nur eine Direktive, nämlich FORWARD (die Direktives in diesem Sinn sind nicht zu verwechseln mit den Compiler-Direktiven wie z.B. {$incl...}!) Direktiven sehen zwar auf den ersten Blick wie Wortsymbole aus, werden aber compilerintern wie Bezeichner (Identifier) behandelt.

Dadurch können sie z.B. vom Programmierer „überdefiniert" werden, d. h. Sie können eine Variable „Forward" oder einen Datantypen „String" nennen. Letzteres wird in der Tat auch häufig in Standard-Pascal-Programmen getan („TYPE String=packed Array[1..xxx] of Char" ist eine sehr beliebte Deklaration). Aus diesem Grunde wurden auch die MaxonPASCAL-typischen Schlüsselwörter wie „STRING" oder „LIBRARY" in Form von Direktiven realisiert.

3.) Konstanten

true
false

MaxInt	= 32767
MaxLong	= 2147483647
MaxLongInt	= 2147483647
MaxCard	= 65535
MaxWord	= 65535
MaxShortCard	= 255
MaxByte	= 255
MaxShort	= 127
MaxShortInt	= 127
Pi	= 3.14159265358979

4.) Typbezeichner

a) Boolean = (false,true)

b) Ganzzahltypen:

Integer	= -32768 .. 32767
ShortInt = Short	= -128 .. 127
LongInt = Long	= -2147483648 .. 2147483647
ShortCard = Byte	= 0 .. 255
Cardinal = Word	= 0 .. 65535

c) Char = chr(0) .. chr(255)

d) Real

e) Text = FILE OF Char

f) Str

g) Ptr

5.) Variablen

Input : text
Output : text
Sysbase: ptr Absolute 4
DosBase: ptr
MathBase: ptr
MathTransBase: ptr
FromWB: boolean
ParameterStr : str
ParameterLen : integer
StartupMessage: ptr
Mem : Array[0..MaxLongInt] Of Byte Absolute 0

6.) Prozeduren

AddExitServer (proc)
Assign (datei,name)
BlockRead (datei, variable, anzahl)
BlockWrite (datei, variable, anzahl)
Buffer (datei, bytes)
CBreak
ClrScr
ClrEol
Close (datei)
CloseConsole (devpointer)
CloseLib (libpointer)
Close_Screen (screenhandle)
Close_Window (windowhandle)
Dec (ordinalvar)
Delay (secs*50)
Delete (stringvar, pos, len)
DelLine
Dispose (pointer)
DisposeAll
Error (string)

Exchange (var1,var2)
Free_Mem (adresse,grösse)
Get (datei)
GotoXY (x,y)
Inc (ordinalvar)
Insert (string,stringvar,pos)
InsLine New (pointer)
OpenLib (pointervar, name, version)
Page [(datei)]
Put (datei)
Randomize
Read (readparameterlist)
ReadLN (readparameterlist)
Reply_Msg (msg pointer)
Reset (datei)
ReWrite (datei)
SetStdIO (consoleptr)
Val (string, numvar, intvar)
Write (writeparameterlist)
WriteLN (writeparameterlist)
WriteCon (devpointer, string)

7.) Funktionen

Abs (numerisch): numerisch
Addr (var): LongInt
Alloc_Mem (größe,flags): LongInt
ArcTan (x): Real
Break (n): Boolean
Chr (int): Char
Concat (str1, str2, ...): String
Copy (string, pos, len): LongInt
Cos (x): Real
Eof [(datei)]: Boolean
EoLN [(datei)]: Boolean

Exp (x): Real
FreeStack: LongInt
Get_Msg (port): Ptr
Hi (i): Integer
IntStr (long): String
Length(string): LongInt
Ln (x): Real
Lo (i): Integer
Odd (int): Boolean
OpenConsole (window): Ptr
Open_Screen (x,y,breite höhe,tiefe,hfarb,vfarb,mode,name): Ptr
Open_Window (x,y,breite,höhe,farbe,IDCMP,flags,name,
screen,minw,minh,maxw,maxh): Ptr
Ord (ordinal): Integer
Pos (str1, str2): LongInt
Pred (ordinal): ordinal
Pwr10 (int): Real
Random: Real
Random (i): Integer
ReadCon (devpointer): Char
RealStr (r,i): String
Round (x): LongInt
Sin (x): Real
SizeOf (typ): LongInt
Sqr (numerisch): numerisch
Sqrt (x): Real
StrLen (string): Integer
Succ (ordinal): ordinal
Swap (i): Integer
Upcase (ch): Char
Wait (maske) : Long
Wait_Port (port): Ptr

Index

$link 196
$path 176
Abbrechen 29
Abhängigkeit 33
Abs 157
ABSOLUTE-Variablen 93
Abspielen 63
AddExitServer 168
AddIDCMP 238
Addr 167
ALink 31, 196
Allocmem 128, 160
AND 115
Anhalten 63
Anspringen 21
Anweisungsteil 104
Arbeitsspeicher 32
ArcTan 157
ARexx-Befehle 39

ARexx-Port 37-38, 81, 284
- addtx 40
- addtxobj 40
- compile 39
- errorscript 41
- exefile 40
- getinfo Make 40
- getinfo STATE 40
- info 39
- leavemk 39
- loadmk 39
- MAXONPASCAL 22
- newmk 39
- objectfile 40
- opcomp 39
- openmake 40
- opsys 39
- Option results 40
- saveconf 40
- savesession 40
- SendAppComm 39
- setopt 40

ARexx-Schnittstelle 37
Argument-String 34
Arrays 130, 139
ASCII-Zeichen 142
Assembler 215
Assemblermodul 34
Assign 143, 151-152
Auflösung wählen 71
Aufnehmen 63
Aufzählung 121, 138
Aufzählungstypen 121
Ausführen 21, 29
Ausschneiden 50
Ausschnittstypen 121, 138
Auszeichnen 60
Auto-Indent 61
Auto-Save 72
Autoren 2
Autosave 35
BACKSPACE 44
BeginIO 247
Benutzerdefinierte Menüs 75
Betriebssystemfunktionen 139
Bezeichner 88, 207
Binärzahlen 115
BLink 31, 196
Block drucken 52
Block einlesen 52

Blockoperationen
- Cut, Copy, Paste 49
- Einrücken 52
- Laden, Speichern, Drucken 51
- Spaltenblöcke 51
- Umwandeln 52

BlockRead 144, 156
BlockWrite 144, 156
Blöcke 89
Bookmarks 60
Boolean 120, 138
Break 164
Buffer 155
Byte 113
Cardinal 113
CASE 108, 135
CBreak 165
Char 121, 138
Chr 157
CLI 43
Close 152
CloseConsole 162
CloseLib 163
Close_Device 247
Close_Screen 162
Close_Window 161
ClrEol 149

ClrScr 149
Compiler-Anweisungen 175
Compiler-Einstellungen 37

Compiler-Fehlermeldungen 56
"Program" expected 255
Absolut variables must get... 249
Boolean Expression expected 249
Branch too far 249
Can not open mathieee... 250
Can not write unit 250
Can´t copy this file 250
Can´t create directories 250
CASE needs an ordinal expression 250
Compiler stack overflow 250
Con not jump into a structure 250
Constant expected 250
DO expected 250
Double definition 250
Double use of Identifier 250
Element type is too large 251
Error in string 251
Error while reading unit 251
Field identifier expected 251
File expected 251
File is not of required Type 251
File must be VAR parameters 251
File not found 251
Filename expected 251
Filename too long 251
FILEOFFILEis not allowed 251
First constant is higher than... 251
FOR needs an ordinal type 251
Functions can return simple 252
Garbage at end of program 252
Global procedure without... 252
Identifier expected 252
Identifier not defined in that... 252
IMPLEMNTATION expected 252
Illegal assignment... 252
Illegal type for textfile input 252
Illegal type for textfile output 252
Incompatible parameterlists 252
Integer expression expected 253
Label expected 253
Label on wrong level 253

Laufzeitfehler
....Division by zero 257
....Error in Case 258
....Error in CloseLib 258
....Can´t open xxx.library 257
....Exec error 258
....Freemem failed 258
....Intuition error 258
....Out of memory 258
....Out of range 257
....Overflow error 258
....Subscript out of range 257
....User Break 258

Lib-base must be of LongInt or ... 253
Magic number does not match 253
Memory overflow 253
Name too long 253
Ne free memory for BSS-Hunk 253
Ne free memory for relocation 253
Nesting overflow 253
No free memory for buffer 253
No free memory for unit 254
Numeric expression expected 254
Offsets must be integer 254
Only global Procedures can... 254
Ordinal Type expected 254
Out of range 254
Overflow at nesting of Incl-files 254
Overflow in floating-point 254
Please set an unit directory 254
Please use „Round" or „Trunc" 254
PROCEDURE or FUNCTION 254
Short Circuit of Units 255
Simple type expected 255
String exceeds line 255
String expression expected 255
String too long 255
Syntax error in compiler directive 255
Temporary string can not get... 255
THEN expected 255
These types can not be converted 255
This function identfier can not 255
This Prcedure can not be used 256
To expected 256
Type conflict 256
Type conflict of Operands 256
Type expected 256
Type identifier expected 256
Type-free pointer must not... 256
Unable to open main file 256
Unexpected end of source 256
Unit not found 256
Unknown identifier or Syntax 257
USES expected 257
Variable expected 257
Variable odd address 257

WITH expects a variable of a... 257
Wrong index type 257

Compiler-Schalter
Arithmetischer Überlauf.....32
Debugger-Datei 33
Indexbereich.....32
Semikolon melden 32
Stackgröße 32
Subrange testen.....32
Unterbrechen 32

Compilermenüs.....24
Compiling 26
Concat 126, 171
Configuration Icon 72
Copy 126, 171
Copyrights und Warenzeichen 2
Cos 157
CreateExtIO 246
CreatePort 246
CreateStdIO 246
Crt 235
CrtConsole.....241
CrtMessage.....239
CrtRastPort.....240
CrtUserPort 241
CrtWindow.....240
Datei 72
Datei speichern 295
Dateien.....151

Dateioperationen
Laden.....49
Speichern.....48

Dateitypen 142
Datentypen 113
DbPwr10.....167
Debugger 34
Dec 171
DEF.....177
Def-Datei laden.....81
Delay.....168
DELETE 44, 172
DeleteExtIO 247
DeletePort.....246
DeleteStdIO 246
DelLine 149
Direktiven 302
Display Beep 73
DISPOSE.....128, 159
DisposeAll 159
DO 107-108
Double.....113, 117
DOWNTO.....107-108
Drucke Zeilennummern 75
EBNF.....85, 104
EBNF-Darstellung.....76
Editor.....42

Editor-Fehlermeldungen
(erwartet 300
) erwartet 301
Alles ist rückgängig gemacht 293
Alles ist wiederhergestellt worden 293
Altes Pictogramm ersetzen 295
Auflösungen nur im Bereich.....298
Automode-Bereich nur von.....298
Cursor steht auf keinem... 292
Datei existiert bereits 295
endif ohne if.....299
Falsches Argument 301
Fehler beim Lesen 297
Funktion würde Einfaltung.....291
Geänderten Text verwerfen.....294
IDENTIFIER erwartet 300
Markierung gesetzt.....290
Kann Datei nicht öffnen.....296
Kann Info-Datei nicht speichern ...297
Kann nicht speichern.....296
Kein Befehl 301
Kein Block definiert 291
Kein Pictogramm vorhanden.....290
Kein Speicher mehr 294, 296
Konnte Backup nicht erzeugen.....291
Konnte Datei nicht speichern 297
Konnte Formular nicht öffnen 297
LISToder IDENTIFIER erwartet.....301
LVAL erwartet.....301
Markierung nicht gesetzt 291
Menü ist zu groß für den Screen...293
NUMBER erwartet 300
Präprocessor-Befehl zu komplex ..299
Präprozessor Befeh erwartet 299
Semikolon erwartet.....301
STRING erwartet 300
Strukturen stimmen nicht überein293
Suchmuster nicht gefunden 292
Tabulatorweite nur zwischen... 298
Text ist geschützt.....290
TITLE erwartet.....299
Wrap-Position darf nicht... 297
Zeichenkete zu lang 292

Zeiten kleiner 30 Sek. machen....298
Zu viele Argumente....301
Zu viele Menüeinträge....299
Zu viele Menütitel....300

Editorkommandos....259
Again....270
AutoMode....283
BackSpace....260
BigStepDown....261
BigStepUp....261
BlockHigher....269
BlockLower....269
BlockSL....269
BlockSR....269
Bot....263
Copy....268
Cut....267
Delete....260
Display....282
Do....282
EditForm....281
EndTextRead....288
EnterCmd....281
ExecCli....289
ExecRexx....284
ExecRexxString....284
Exit....279
Find....270
FindForm....280
FindRepForm....280
Fold....272
Font....283
FreeText....264
FreeWin....278
FullSizeWin....279
GetBlockInfo....286
GetLineString....284
GetScreenInfo....287
GetTextInfo....286
GetTextList....285
GetTextPos....285
GetWindowList....285
GetWord....284
GlobalForm....280
GoBookMark....274
GoDispPos....263
GoMatchStruct....271
GoParseError....271
GoPredFold....273
GoPredKey....273
GoStringPos....263
GoSuccFold....273
GoSuccKey....273
GotoForm....280
Iconify....289
IndentReturn....261
InfoForm....281
InitTextRead....287
InsertChars....289
InsertFile....266
Jump....271
KillLine....260
Mark....267
MarkVert....267
ModifyOldCmd....282
MoveDown....262
MoveLeft....262
MoveRight....262
MoveUp....261
New....265
NewText....264
NewWin....278
Open....266
OpenClip....268
OpenDefs....283
PageForm....280
Paste....268
PasteVert....268
PlayRecScript....277
PredText....264
PredWin....278
Print....265
Quit....279
RecPlay....277
RecStart....277
RecStop....277
ReDo....288
RemFold....272
RemTab....271
RepeatCmd....281
Replace....270
ReplaceAll....270
Return....261
Save....266
SaveClip....268
SaveRecScript....277
ScreenDown....262
ScreenUp....262
SelTextN....264
SendAppComm....23
SetBookMark....274
SetFoldsOff....272
SetFoldsOn....272

SetMode 289
SetOverOff 275
SetOverOn 275
SetTab 275
SetWrap 275
SetWrapOff 276
SetWrapOn 275
ShowFold 272
SplitHorWin 279
SplitVerWin 279
SuccText 264
SuccWin 278
TextRead 288
ToggleKey 273
Top 263
UnDo 288
UndoLine 260
UpdateText 266
UpdateTexts 267
WrapPara 276
WrapText 276

Edward 42
EdwardCommand 77
Ein-/Ausgabefehler 185
Einfügen 50
Einstellungen 70
ELSE 106, 109, 177
EmptyLN 150
Ende 69
ENDIF 177
Entferne Falte 59
EoF 150
Eoln 150
Error 168, 178
Ersetzen 55
Erstelle Backup 73
Erzeuge Falte 59
Exchange 170
Exe-Datei 33
Exit 169
EXP 119, 157
Exponent 113
Exponentialdarstellung 119
EXPORT 204
EXTERNAL 204
Externer Assembler 34
Fallunterscheidung 106
False 120
Falten 57
Falten überspringen 75
Farben 71
Fatal Error 26
Fehlerdateien 34
Fehlermeldungen 41
Fehlerstellenansprung 41

Fenster
Erzeugen 65
Mit Ansichten arbeiten 67
Schließen 66
Wählen 66

Fensterdaten 36
Fertig 26
FILE OF Char 142
Filehandle 153
Filepos 154
Filesize 154
Fließkomma-Coprozessor 117
Fließpunktzahlen 113
Fontgröße 71
Fontname 71
FOR 107
Form Feed 75
FORWARD 103, 204
FPU 117
Frac 167
Freemem 128
FreeStack 165
Free_Mem 161
FromWB 145
FUNCTION 101
Funktionen 93, 101, 147, 305
Fuß 74
Ganzzahl 138
Ganzzahltyp 113
Gehe zu Fehler 56
Gehe zur Zeile 57
Genauigkeit 119
Get 143, 152
Get_Msg 164
Gleitpunktzahlen 117
GOTO 111
GotoXY 149
Großbuchstaben 53
GROSS/klein 31
Halt 168
Hauptdatei 31
Hi 166
Hochkomma 121
HotKey 69, 81
Hunks 197
Iconify 69

Icons erzeugen 33
IF 106, 177
IMPLEMENTATION 204
IMPORT 204
Inc 170
Incl 175
Include 175
Include-Files 188
Includefiles 175
Indexbereich 181
Information 30
Initialisierung 209
Input 145
Insert 172
InsLine 149
Installation 18
Integer 113
Interaktiv 221
IntStr 173
IOResult 143, 153
ISO-Standard 102, 109, 140, 151
KeyDef 79
KeyPressed 235
Keys 79
Klammern Einzug 62
Kleinbuchstaben 53
Kommandozeilen-Interpreter 70
Kommentare 88
Konfigurationsdatei 35
Konstanten 90, 145, 303
Kopf 74
Kopieren 50
Label 90
Laden 75
Laufzeit-Protokoll 34
Laufzeitfehler 257
Length 172
Lesezeichen 60
Libraries 186
Linker Error 26
Linker Rand 75
Linkersteuerung 179, 194
Linking 26
Literale 139
Lizenzbedingungen 3
LN 119, 158
Lo 166
Long 113
LongInt 113
LongReal 113, 117
Main File 31
Make 31, 39, 196, 219
Make-Units 31
Makros 63
Mantisse 113
Markieren 50
MathBase 145
MathtransBase 145
Mem 146
Menu 77
MenuGroup 77
MenuItem 78
MessageReceived 236
ModeQualList 79
Module 191, 205
Näherungswert 119
NameOfProgram 241
Neuer Font 71
NEW 128, 159
NewList 246
NewWindow 139
NOT 115
Numerische Typen 113
Nummernbereich 75
Objektdatei 33
Objektdateien 194
Odd 158
OpenConsole 162
OpenLib 163
OpenScreen 129
OpenWindow 139
Open_Device 247
Open_Screen 161
Open_Window 160
Optionen 179
Optionen sichern 35
OR 115
Ord 158
OTHERWISE 109
Out of Memory 26
Output 145
Page 149
Parameter 97
ParameterLen 34, 145
ParameterStr 34, 145
Pascal-Schlüsselwörter 302
PASCALCrt 235
Path 176
Pfad 73
Pointer 159
Pointertypen 128
Pos 171
Pred 158
PROCEDURE 101

Programmkopf ... 89
Programmstart ... 28
Projekt & Ende ... 68
Projekte ... 212
Projekte erstellen ... 221

Projektverwaltung ... 219
- Abhängigkeiten ... 225
- Datei wechseln ... 227
- Dateien hinzufügen ... 224
- Externe Übersetzung ... 230
- Handgemachte Makedateien ... 228
- Interne Übersetzung ... 227
- Internes Compilieren ... 229
- Kommentare ... 231
- Linkeraufruf ... 228
- Make ... 22
- makefile ... 22
- Makros ... 231
- Projektpflege ... 226
- Regeln ... 232
- Übersetzen ... 224
- Übersetzung erzwingen ... 226
- Zeilenverbindung ... 230

Prozeduraufruf ... 105
Prozeduren ... 93, 147, 304
Prozedurparameter ... 102
Prüfsummen ... 210
Pseudo-Bezeichner ... 101
Put ... 143, 152
Pwr10 ... 167
Quelltext-Veränderungen ... 28
QuotedString ... 77
Random ... 165
Randomize ... 166
Read ... 117, 148
ReadCon ... 162
ReadKey ... 235-236
ReadLn ... 149
Real ... 113
RealStr ... 173
Realzahlen ... 113
Rechengenauigkeit ... 113, 118
Rechter Rand ... 62, 75
RECORD-Typ ... 110
Records ... 130, 139
Redo ... 82
REPEAT ... 107
Reply_Msg ... 164
Reset ... 143, 151
Rewrite ... 143, 152
Round ... 158
Rundungsfehler ... 119
Schachtelung ... 206
Scrollbereich ... 73
Seek ... 143, 154
SeekEoF ... 150
SeekEoLN ... 150
Seitenformat ... 74
Semikolon ... 88
SET ... 129
SetIDCMP ... 238
SetMode ... 79
SetStdIO ... 162, 235
SHL ... 115
Short ... 113
Short-Cuts ... 70
ShortCard ... 113
ShortInt ... 113
SHR ... 115
SIN ... 119, 158
SizeOf ... 168
Speichern ... 75
Spezialsymbole ... 87
Sqr ... 158
SQRT ... 119, 159
Stacküberlauf ... 182
Standard-Units ... 235
StartupMessage ... 145
STATIC ... 93, 182
Status-Anzeige ... 43
String-Konstanten ... 126
Stringbehandlung ... 171
Stringoptionen ... 36
Stringtypen ... 123
StrLen ... 172
Struktur-Check ... 55
Strukturieren ... 57
Strukturkontrolle ... 56
SubIDCMP ... 239
Subrange ... 180
Succ ... 159
Suchen ... 53, 55
Support ... 2
Swap ... 166
Symbole ... 87
Syntax ... 85, 104, 192
System ... 30
Systemeinstellungen ... 33
TAB in Spaces ... 61
TAB Weite ... 62
Temporärer String ... 126

Text
Auswahl 46
Drucken 47
Erzeugen 46
Freigeben 47
Löschen 47

TextColms 241
Textdraw 73
TextLines 241
THEN 106
Titelzeile 43
TO 108
True 120
Trunc 159
Typbezeichner 303
Typdefinition 91
Typkonvertierungen 136
Überladung 118
Überschreiben 61
Übersetze CRs 62
Übersetzen 20
Undefined Type 256
Undo 82
Undo Konfiguration 72
UndoLine 44
Unit 203
Units 191, 201, 205
Unterbrechen 29, 181
UNTIL 107
Upcase 166
USES 203
Val 174
Variablen 145, 198, 304
Variablendeklaration 92
Variablenparameter 98
Variante 135
Verbundanweisung 106
Verbundtyp 131
Vertauschen 55
Vertikal einfügen 51
Vertikal markieren 51
Vorwort 17
Wait 163
WaitForKey 236
Wait_Port 163
Wertzuweisung 105
WhereX 241
WhereY 241
WHILE 107
Wildcard 54
WindowClosed 239
WindowTitles 241
WITH 110
Word 113
Word Wrap 61
Word-Wrap 62
Workspace 32
Workspace overflow 26
Wortsymbole 87
Write 117, 147
WriteCon 162
WriteLn 148
XOR 115
Zeichenkette 126
Zeige EOL 62
Zeige Falte 59
Zeige Spaces 62
Zeige TABS 61
Zeilen bis zum Text 74
Zeilen nach dem Text 74
Zeilen/Seite 74
Zeilen/Text 75
Zeit 73
Zum Anfang 74
Zum Ende 75

Maxon PASCAL 3

Features

Compiler
- sehr schnelle Turnaroundzeiten
- komfortable Projektverwaltung (Make)
- direkter Fehlerstellenansprung
- voll in den Editor integriert
- über ARexx-Schnittstelle steuerbar
- Includes/Units für AMIGA-OS 3.1
- speicherbarer Programmstatus
- kompilierte Programme sind auf allen Amiga lauffähig (A500 - A4000)
- linkfähig: Objektdateien anderer Sprachen wie z.B. Assembler, C/C++, MaxonBASIC oder Modula können dazugebunden werden
- weitgehend kompatibel zu Borland Turbo Pascal 5.0

Editor (EDWARD)
- Multi-Window-Technik
- beliebig viele Texte gleichzeitig bearbeitbar (speicherabhängig)
- Falten von Textpassagen
- Tastatur und Menüfunktionen frei belegbar
- definierbare Makros
- umfangreiche UNDO-Möglichkeiten
- Autosave- und Backup-Modus
- speicherbarer Programmstatus
- ARexx-Schnittstelle

Bibliotheken
- CRT-UNIT (eigene Konsole, KeyPressed, ReadKey, WaitForKey)
- PASCALCrt-UNIT (im Source)
- SYSTEM-UNIT (Unit der System-Include-Dateien des Amiga mit allen Deklarationen von Konstanten, Datentypen und Library-Funktionen für den vollen Zugriff auf das Betriebssystem)

Der neue PASCAL-Compiler

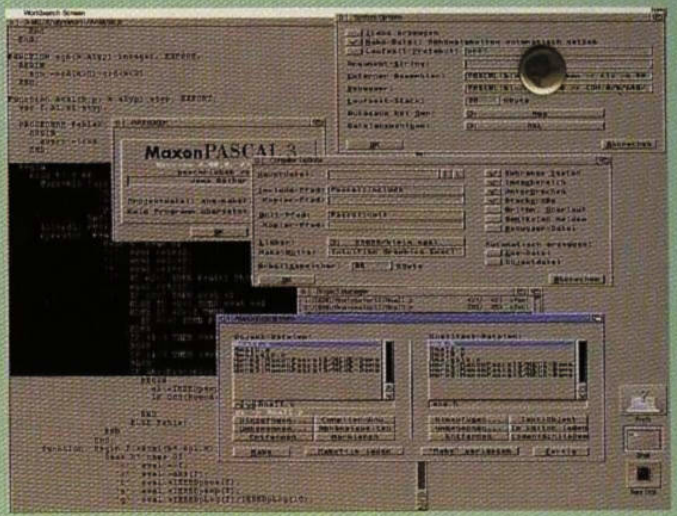
MaxonPASCAL baut auf dem Jensen/Wirth Standard auf, ist jedoch um sehr viele leistungsfähige Funktionen und Prozeduren erweitert worden. Dies betrifft in erster Linie Datei-, String- und AMIGA-typische Funktionen.

Compiler/Linker

Compiler und Linker sind voll in das System integriert. Dadurch entfallen sämtliche Nachladezeiten. Die Möglichkeit, alle benötigten Include-Dateien beim ersten Compiler-Lauf in die RAM-Disk zu kopieren, erhöht die Turnaround-Zeiten enorm. Im Schnitt compiliert das System 20.000 Zeilen pro Minute.

Units

Eine der wichtigsten Möglichkeiten von MaxonPASCAL ist das UNIT-Konzept. Es unterstützt den modularen Aufbau von Programmen durch Auslagern von Routinen. Die einzelnen Quelltexte Ihrer Programme werden dadurch kürzer und übersichtlicher. Fertige Units werden vorcompiliert und bei weiteren Compilerläufen nur noch dazugelinkt. Dies bringt einen enormen Geschwindigkeitsgewinn. Auch die AMIGA-System-Includes zu AMIGA OS 3.1 werden als Units mitgeliefert.



Einsteiger und Umsteiger

MaxonPASCAL ist das ideale System für Einsteiger und Umsteiger. Zum einen ist PASCAL die ideale Sprache um das Programmieren zu erlernen und zum anderen ist MaxonPASCAL der Compiler, der durch seine extrem einfache Bedienung den Ein- oder Umstieg sehr leicht macht.

Includes für OS 3.1

MaxonPASCAL bietet zudem den einfachen Zugriff auf die AMIGA-Betriebssystemfunktionen aller Versionen bis zu OS 3.1. Die entsprechenden Includes und Units werden mitgeliefert. Damit ist es jedem möglich, den AMIGA unter allen verfügbaren Betriebssystemen wie ein Profi zu programmieren.

Erweiterbar

MaxonPASCAL ist ein offenes System, das Sie nach Belieben erweitern können. Mögliche Zusätze sind z.B. das Hilfesystem HOTHELP 3, der Assembler MaxonASM oder MakeAPP, das Werkzeug zum mausgesteuerten Erstellen von Programmoberflächen.

Die Systemvoraussetzungen sind ein AMIGA (A500- A4000) mit mindestens 2 MB RAM, AMIGA-OS 1.3 bis 3.1 und zwei Diskettenlaufwerken. Ein Festplattenlaufwerk wird empfohlen.

AMIGA®

4 010509 411401

MAXON Computer GmbH • Industriestr. 26
D-65760 Eschborn • Telefon 06196/481811
Irrtümer und Änderungen vorbehalten!

MAXON computer

FUTURE